Metamathematics

Jacob Denson

March 16, 2017

Table Of Contents

Part I

Mathematical Logic

Chapter 1

What's Logic All About?

In Mathematics, we use rigorous arguments, known as proofs, to discover new facts from prior assumptions. Metamathematics turns mathematics on its head, using the mathematical method to analyze mathematical methods themselves! Traditionally, metamathematics (or logic, as it was then called) was used by philosophers to find valid forms of reasoning in arguments. But in the early 20th century, a desire for rigour and an invention of an incredibly powerful system of mathematics unvealed a plague of foundational mathematical paradoxes in forms of reasoning believed to be valid. It was in this firestorm where true metamathematics was forged...

In the late 1800s, a German mathematician named Georg Cantor invented an entirely new way of thinking about mathematics. Instead of discussing particular numbers or shapes or functions, Cantor decided instead to talk about collections of mathematical objects, known as sets, and the ways these sets can be manipulated and described. In a flourish, he solved a problem which had troubled mathematicians for centuries. A number is called *algebraic* if it is the root of a polynomial

$$a_0 + a_1 X + \cdots + a_n X^n$$

with rational coefficients $a_0, \dots, a_n \in \mathbf{Q}$. Since the time of Archimedes, mathematicians had wondered whether all numbers were algebraic. Cantor answered this question in the negative. Rather than finding an example of a trancendental number, he counted the *set* of algebraic numbers, counted the *set* of all numbers, and proved that there were far too many numbers for them all to be algebraic. But by creating a theory powerful enough to collect mathematical concepts together, and providing a linguistic framework which changed the way mathematicians thought about mathematics, Cantor opened the door for mathematicians to encapsulated the entirety of mathematics, and from this doorway paradoxes emerged.

Example (Cantor's Paradox)**.** *For any set X, we can consider the powerset 2^X, which consists of all sets Y which are subsets of X. Cantor proved that the cardinality of 2^X is always strictly greater than the cardinality of X. But if X is the set of all mathematical objects (including sets, and sets and sets, and so on and so forth), then $2^X \subset X$, which implies the cardinality of 2^X is less than or equal to the cardinality of X, contradicting Cantor's theorem.*

Example (Burali-Forti Paradox)**.** *Consider the set Ω of all ordinals. Then Ω is a well-ordered set, and hence is order isomorphism to some ordinal, and thus to a proper segment of itself. But no well-ordered set is order isomorphic to a proper subset of itself!*

Example (Russell's Paradox)**.** *Consider the following set.*

$$X = \{x : x \notin x\}$$

Russell's paradox rests on an innocent question: Is $X \in X$? If X was in X, then by definition, we conclude that X cannot be contained in itself. So we are lead to believe that $X \notin X$. But then, by construction of X, we conclude X is in X after all! A more colloquial explanation of the paradox considers a town with a single barber, shaving everyone who does not shave themselves. Russell's paradox occurs when we ask if the barber shaves himself?

All of these paradoxes rely on the fact that set theory can discuss mathematical objects that are 'too big' to be understood by colloquial forms of logic. Russell's paradox is the most mathematically elegant of the set-theoretic paradoxes, for it relies on no advanced knowledge of cardinals nor ordinals. However, the paradox is essentially equivalent to Cantor's theorem. The proof of this theorem takes an arbitrary $f : X \to 2^X$, lets $Y = \{x \in X : x \notin f(x)\}$, and, assuming that f is surjective, considers y with $f(y) = Y$, in which we obtain that $y \in Y$ if and only if $y \notin Y$. If we replace an arbitrary set X with the set of all sets, then $2^X = X$, so if we take the identity function as f, then $Y = \{x \in X : x \notin x\}$, which is exactly the set involved in Russell's paradox.

Henri Poincaré saw the paradoxes as proof that set theory was a "plague on mathematics". But paradoxes were not limited to set-theoretic concepts. In their work on the Continuum hypothesis, Julius König and Jules Richard found paradoxes attacking the notion of mathematical definability, the ability to specify objects by phrases of english.

Example (Richard & König's Paradoxes)**.** *The set X of all english expressions is countable, because there are only finitely many expressions of a certain length, and the union of countably many disjoint finite sets is countable. A certain subset Y of X correspond to expressions describing real numbers, and we can consider an interpretation of these expressions as a function $f : Y \to \mathbf{R}$. Since $\mathbf{R}$ is uncountable, there are some real numbers which cannot be described in english. We call the numbers in $f(Y)$ the* **definable real numbers***. As the image of a countable set, the set of all definable real numbers can be ordered, and we can construct the expression S, which is*

"The number constructed from Cantor's diagonal argument on the enumeration considered."

Then we would expect that S defines a real number, and therefore $f(S)$ is a definable real number, yet by our understanding of Cantor's diagonal argument we also conclude that, if the semantics of the english language is consistant, that $f(S) \notin f(Y)$. Similarily, if we let ω be some well-ordering of the real numbers, then we could consider the expression T, which is

"The least number not definable in english with respect to ω."

and then T should describe some definable number $f(T)$, which by our understanding of the semantics of english phrases, we would expect $f(T)$ not to be definable.

It seems obvious that a definition is simply a description of the qualities of an object under consideration, but if this were true, there would be no problem with the arguments above, so we are at an impasse. A precise definition of definability is a key discovery in metamathematics, from which we will obtain the beautiful results of Gödel and Tarski. A related class of paradoxex results from self-reference.

Example (Löb)**.** *Consider the Preposition B, defined to be true when $B \implies A$ is true. If B is true, then $B \implies A$ is true, so A is true. But then $B \implies A$ is true by construction, so B is true, and we conclude A is true. Since A was arbitrary, we can conclude that every logical statement is true!*

It can be argued that Löb's paradox fails because self referential statements are naturally circular, but Curry showed that self reference occurs much more subtly in set theory.

Example (Curry). *For any property P, consider*

$$C = \{x : (x \in x) \implies P(x)\}$$

Then $(C \in C)$ holds if and only if $(C \in C) \implies P(C)$ holds. We must have $C \in C$, for if $C \notin C$, then $(C \in C) \implies P(C)$ holds vacantly. But this implies $(C \in C) \implies P(C)$, so $P(C)$ is true. We conclude $P(C)$ is true, irrespective of the content of the statement $P(C)$.

It became clear to the mathematicians of the early 1900s that the current state of logic was not shapr enough to fix or explain away the paradoxes of set theory; our definitions had not been analyzed in enough detail for them to handle the current state of mathematics. Thus the field of 'modern' metamathematics was formed. Our only hope is to apply the precise weapon of mathematical rigor. In the face of adversity, we do what mathematicians do best – define and conquer.

1.1 Formal Systems

In order to analyze mathematics mathematically, a careful method must be employed to avoid circular reasoning. The main trick is to construct abstract models which simulate the procedure of mathematics, while being clearly separated from the mathematical procedures we use to rigourously understand these models. Mathematical logic is effectively a form of mathematical linguistics – the study of the languages that a mathematicians uses in reasoning. To do this, we must first estimate the procedure of of a mathematician at work. First, she accepts some fundamental statements as 'obviously true', known as axioms. Using accepted logical derivations, additional statements are obtained from the axioms. Mathematics is as simple as that. Thus all we need to formalize is what a *statement is*, and what a *logical derivation* is.

We emphasize that the choice of axioms is completely arbitrary. They can be determined individually by the person who is performing a logical deduction. We would hope our logical analysis holds irrespective of the axioms, and since virtually no statements are accepted as universally true, our logical analysis can assume very little about what is assumed. We assume the choice of axioms is determined by the person at hand. To a scientist, what is obvious is that determined by experiment. To a priest, what is obvious is that which is written in a holy book. In mathematics, we do not care how one determines that statements are obvious, only that some statements are accepted as obvious, and of the statements we are interested in, we are able to clearly separate which are accepted and which are not accepted, so we can proceed apriori.

The basic object of study in metamathematics, from which we build all of our models, is deceptively simple. We take a set Λ, known as an **alphabet**, and consider **strings** over that alphabet, finite (possibly empty) sequences of elements in Λ. Rather than formalizing the thought process of a mathematician, we choose instead to formalize what is written down: the proof. Abstract strings have turned out to be the right formalization to understand this approach, and we shall find they suffice to express all of finitary mathematics.

We will often denote a string $(v_1, \dots, v_n)$ as $v_1 \dots v_n$. When this might confuse the reader, we surround the string in quotation marks, like "$v_1 \dots v_n$". However, the choice of alphabet, visually distinct from the Latin alphabet, will often make the quotation marks redundant. We shall identify an element in Λ with the corresponding one letter string in Λ^*. The concatenation sw of two strings $s = s_1 \dots s_n$ and $w = w_1 \dots w_m$, is the string $s_1 \dots s_n w_1 \dots w_m$. If we view concatenation as an associative algebraic operation, then Λ^* is the smallest *monoid* containing Λ. Since most of our languages will be constructed from putting basic strings together, we may wish to take complicated strings, such as

"It was the best of times, it was the worst of times"

and identify more basic substrings in them, such as "worst". A **substring** of a string s is a string u such that $s = wuv$. Substrings are consecutive subsequences of characters in s. A **language** is a subset of strings over an alphabet. It allows us to separate meaningful strings over the alphabet

"The quick brown fox jumps over the lazy dog"

from meaningless garble, like

"iovuiaesfpauaauupewbpvapbuib"

In mathematical logic, the languages of study consist of the set of meaningful formulae in some logical calculus, and various sublanguage of these formulae, whether they be the true formulas, the provable formulaes, or the satisfiable formulas. The construction of such 'truthful formulae' is encapsulated in what is called a **formal system**. The most general definition consists of a language L over an alphabet Λ, a set of axioms, which form a subset of L, and a set of inference rules, pairs (Γ, s) where $\Gamma \subset L$ is the premises of the inference, and s is the conclusion. The **theorems** of a formal system are members of the smallest set such that

1. every axiom is a theorem

2. if (Γ, s) is an inference rule, and all elements of Γ are theorems, then the formula s is a theorem.

If F is a formal system, we shall let $\vdash_F s$ state that s is a theorem of F. Normally though, we will write $\vdash s$, for the formal system is clear from context. Just as algebra is the study of groups, rings, and fields, metamathematics is the exploration of the different formal systems we can use to model mathematics.

1.2 Why Trust Mathematical Logic?

Before we get to the real work though, how can we ensure the models we apply to analyze mathematics are robust enough to tell us about real mathematics? David Hilbert's plan, along with the rest of the formalist school, was to construct a formal system powerful enough that it could discuss itself, and prove itself consistant (without paradox). If such a system could be constructed, Hilbert believed we could hide the rest of mathematics in this system, shielding mathematics from paradoxes. One reduces mathematics to abstract symbol pushing inside the system, but Hilbert did not see this as an issue; this symbol pushing is no different from the symbol pushing inside our minds when we solve a problem, albeit more explicit. Regardless of whether you believe in this approach, we shall find Hilbert's approach is doomed from the beginning, for no sufficiently advanced consistant formal system can prove itself consistant.

So how can we ensure that our formal systems give correct results about everyday mathematics? If you desire absolute facts, you will be dissapointed. We can never hode to model a real life situation completely accurately. A physicists's models are ideals, carved from reality in all senses but experimental parameters. No model describes a systems evolution exactly, and it is myopic to suggest a model's perfection. In spite of this, physics still does a bloody good job! In metamathematics, we attempt to form a mathematical model of mathematical principles. Some principles are pinned down for examination, others lost. We hope this model has enough vitality to provide key insights into real-life mathematics. Whether the method is successful can only be determined by the correspondence between the results of metamathematics, and evidence in actual mathematics.

A source of confusion in physics is the stylistic treatment of assumptions as absolute facts. A physicist describes "a planet moving according to the equation $\ddot{x} = -m/x^2$" even if he is actually talking about the dynamical system whose evolution is described by the differential equation $\ddot{x} = -m/x^2$, which *models* the motion of a planet. Such expressions are unavoidable, since they make the study of mechanics much more viceral and appealing to intuition, whereas eschewing the natural language makes the formal equivalent dry to the bone. Keep this principle in mind as we begin to build models of logic. Every theorem we prove must be judged for authenticity outside of the mathematical model we have created.

Chapter 2

Prepositional Logic

We shall begin with propositional logic, the simplest formal system to analyze truth. To understand propositional logic, we construct a mathematical model, known as a formal language, which represents the language in which mathematics is performed. The formal language is then analyzed by common mathematical deduction rules. The standard formal language for logic is an analysis of strings, sequences of abstract symbols from a given alphabet. Strings represent mathematical statements; manipulating these strings models how a mathematician infers some mathematical statement from another. It is best to see the tool in action to understand its utility, so we proceed swiftly into the technicalities involved in the construction of the logic.

2.1 Syntax

Normally, a formal system makes colloquial speech, so each symbol in the alphabet provides a precise representation of some forms of colloquial speech. We begin with prepositional logic, which models statements with sentences of mathematics which are composed of basic statements which are true or false, and independent of one another. Some statements are **atomic** in the propositional logic, because they cannot be divided into more base statements. "Socrates is a man" is an atomic statement, as is "every woman is human". "Socrates is a man and every woman is a human" is not atomic, for the statement consists of two separate statements, composed by the connective "and". In English, "every woman is a human" can be broken into statements such as "Julie is a human" and "Laura is a human", yet propositional logic still considers this statement as atomic; the model does not have the capability to precisely model understanding of these complex statements, which are the realm of predicate logic, discussed in the next chapter.

Let Λ be a set disjoint from $\{(,), \wedge, \vee, \neg, \Rightarrow, \Leftrightarrow\}$. These symbols will represent the atomic statements in our representation of propositional logic. The **propositional language with atoms in** Λ, denoted $\text{SL}(\Lambda)$, is the smallest subset of $(\Lambda \cup \{(,), \wedge, \vee, \neg, \Rightarrow, \Leftrightarrow\})^*$ such that

1. $\Lambda \subset \text{SL}(\Lambda)$.

2. If $s, w \in \text{SL}(\Lambda)$, then $(\neg s), (s \wedge w), (s \vee w), (s \Rightarrow w), (s \Leftrightarrow w) \in \text{SL}(\Lambda)$.

An element of $\text{SL}(\Lambda)$ is called a **formula** or **statement**.

Each **connective** of prepositional logic represents a certain linguistical form. Later on, the connection of symbols to meaning will become clear. For now, they are abstract symbols without intrinsic meaning. Viewing a formal system as meaningless symbol shifting is known as the *syntactical view* of a formal system.

Connective	Name of Connective	Meaning of statement
$\neg s$	Negation	"*s* is *not* true"
$s \wedge w$	Conjuction	"*s and w* is true"
$s \vee w$	Disjunction	"*s or w* is true"
$s \Rightarrow w$	Implication	"*If s* is true, *then w* is true"
$s \Leftrightarrow w$	Bicondition	"*s* is true, *if, and only if, w* is true"

Take care to notice that $\text{SL}(\Lambda)$ is the *smallest* set constructed, in the same way that most 'smallest objects' exist in mathematics, because the intersection of sets satisfying the set of statements defining the set also satisfy the statements. This property leads to the most useful proof method in logic.

Theorem 2.1 (Structural Induction)**.** *Consider a proposition that can be applied to $SL(\Lambda)$. Suppose the proposition is true of all elements of Λ, and that if the proposition is true of s and w, then the proposition is also true of $(\neg s), (s \wedge w), (s \vee w), (s \Rightarrow w)$, and $(s \Leftrightarrow w)$. Then the proposition is true for all of $SL(\Lambda)$.*

Proof. Let P be some property to consider. Note the set

$$K = \{s \in (\Lambda \cup \{(,), \wedge, \vee, \neg, \Rightarrow, \Leftrightarrow\})^* : P(s) \text{ is true}\}$$

satisfies axioms (1) and (2) which define SL(Λ), so SL(Λ) $\subset K$. $\square$

The 'formulas' of prepositional logic are just abstract sequences of symbols. They have no intrinsic grammatical structure. In the way we have defined it, the grammatical structure which appears obvious must be proved. Since we are working over formulas of arbitrary complexity, structural induction will be the most useful here.

Theorem 2.2. *Any sentence in* $SL(\Lambda)$ *contains as many left as right brackets.*

Proof. Any atom in Λ contains no left brackets, and no right brackets, and thus the same number of each. Let s have n pairs of brackets, and let w have m. Then $(\neg s)$ contains $n + 1$ pairs of brackets, and $(s \circ w)$, where $\circ \in \{\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow\}$, contains $n + m + 1$ brackets. By structural induction, we have proved our claim. $\square$

We need a more in depth theorem to correctly parse statements. If statements can be parsed in two different ways, they become ambiguous. For instance, what is the value of $2 + 7 - 5 - 4$? Is it, by collecting factors in pairs, $9 - 1 = 8$, or $9 - 9 = 0$. This is the reason for parenthesis. One splits mathematical logic into syntax, studying strings, and semantics, interpreting strings. We are studying syntax to begin with, so that we may begin semantics. Equations must be understood before we calculate with them.

Theorem 2.3. *If* $wu = s \in SL(\Lambda)$, *where* $w \neq \varepsilon$, *then* w *has at least as many left brackets as right brackets, and* $w = s$ *if and only if* w *has the same number of left and right brackets.*

Proof. If $s \in \Lambda$, then s is only one letter long, so $s = w$, and has no brackets. Now let s and s' satisfy the theorem. We split our proof into two cases.

1. $wu = (\neg s)$: March through all cases. Suppose w has the same number of left brackets than right. w cannot equal (or $(\neg$, nor $(\neg v$, where v is a prefix of s; by induction, v has at least as many left brackets as right brackets, and then w has more left brackets than right brackets. Thus w must equal $(\neg s)$.

2. $wu = (s \circ s')$, for some $\circ \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$: Continue the string march. w cannot equal (, nor $(v$ by induction. Similarily, w cannot equal $(s\circ$, nor $(s \circ v$, where v is a substring of s', so w must equal $(s \circ s')$.

Careful analysis of each case also shows that w must have at least as many left brackets as right brackets. ☐

Corollary 2.4. *Every string in $SL(\Lambda)$ can be written uniquely as an atom Λ, or $(\neg s)$ and $(s \circ w)$, where s and w are elements of $SL(\Lambda)$. The unique connective in the representative is known as the* **principal connective** *of the statement.*

Proof. Such representations trivially exist. Suppose we have two representations. If one of the representations is an element of Λ, the other representation must have length one, and is therefore equal to the other representation. If we have two representations $(\neg s) = (\neg w)$, then by chopping off symbols, $s = w$. It is impossible to have two distinct representations $(s \circ w) = (\neg u)$, for no element of $\mathrm{SL}(\Lambda)$ begins with $\neg$. Finally, suppose we have two representations $(s \circ w) = (u \circ v)$. Then either u is a substring of s, or s is a substring of u, and one is the prefix of the other. But both have balanced brackets, which implies $s = u$, and by chopping letters away, $w = v$. Thus representations are unique. ☐

Given an arbitrary language with a recursive construction, it is in general very difficult to verify if a language is ambiguous. This manifests in the theory of computability, where we find there is no general algorithm with which we can prove a language is ambiguous. However, the particular languages we use to represent formal systems tend to have a fairly simple syntax, and therefore we can prove that these languages are ambiguous with relative ease.

Because we have unique parsing, we can define functions on terms of propositional logics recursively, without having to worry whether such a function is well defined. For instance, let $\Lambda = \{A, B, C\}$, and $\Gamma = \{X, Y, Z\}$. We would like to consider $\mathrm{SL}(\Lambda)$ and $\mathrm{SL}(\Gamma)$ the same, by considering a natural bijection. We extend the map

$$X \mapsto A \quad Y \mapsto B \quad Z \mapsto C$$

by the definition

$$f((s \circ w)) = (f(s) \circ f(w)) \quad f((\neg s)) = (\neg f(s))$$

Such a map is well defined on all of $\mathrm{SL}(\Gamma)$ by the corollary, and injective, and it is easy to see that this map preserves the semantic properties of propositional logic we will soon define.

It is useful to have a visual specification of the decomposition of formulae. Given a formula s, define the **parse tree** of s inductively:

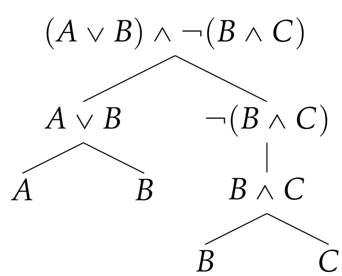
1. If s is atomic, then the parse tree of s consists of a single node, s itself.

2. If $s = w \circ u$, where $\circ$ is a binary connective, then the parse tree has a parent node s, descending into two subtrees, the first of which being the parse tree of w, and the second of which the parse tree for u.

3. If $s = \neg w$, then the parse tree has a parent node s, with a single edge descending into the parse tree of w.

the **complexity** of a formula is the height of the parse tree. A **subformula** of a formula s is a formula associated to one of the nodes in the parse tree of s. It is easy to see that they are the only substrings of s which are valid formulas in the language. An occurence of a formula w in a formula s is a node in the parse tree of s whose associated string is w.

Example. *The parse tree for* $(A \vee B) \wedge \neg(B \wedge C)$ *is*



Thus the formula has complexity 3, *for the longest branch in the parse tree is three. The subformulas of the string are simply the nodes in the tree. The tree also tells us that there are two occurences of B in s, whereas only one occurence of* $(B \wedge C)$.

Before we finish with syntax, it is interesting to discuss a less natural, but syntactically simpler method of forming sentences, called **polish notation**, after its inventor Jan Łukasiewicz. Rather than writing connectives in **infix notation**, like $(u \wedge v)$ and $(u \Rightarrow v)$, we use **prefix notation**, like $\wedge uv$ and $\Rightarrow uv$. We do not need brackets to parse statements anymore, but it is much more easy to read the statement "$a \Rightarrow ((b \vee c) \wedge d)$", than "$\wedge \Rightarrow a \vee bcd$". Nonetheless, the languages are effectively the same, as can be seen as taking parse trees of corresponding formulae in the different languages.

2.2 Semantics

We can understand the discussion in the last section without any understanding of what symbols mean. Now we want to interpret the symbols, giving the symbols meaning. A basic semantic method is to define whether a statement is 'true'. A **truth assignment** on a set Λ is a map $f : \Lambda \to \{\top, \bot\}$.

Example. *An n-ary* **boolean function** *is a truth assignment, where* $\Lambda = \{\top, \bot\}^n$. *It is common to define such functions by truth tables. For n-ary boolean functions, we form a table with* $n+1$ *columns, and* 2^n *rows. In each row, we fill in a particular truth assignment in* $\{\top, \bot\}^n$, *and in the last column, the value of image of the truth assignment under* f. *One may combine multiple n-ary truth functions into the same table for brevity. There are* 2^{2^n} *n-ary truth functions.*

x	y	$H_\wedge(x,y)$	$H_\vee(x,y)$	$H_\Rightarrow(x,y)$	$H_\Leftrightarrow(x,y)$	$H_\neg(x)$
$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\top$	$\top$
$\bot$	$\top$	$\bot$	$\top$	$\top$	$\bot$	$\top$
$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\bot$
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\bot$

This table defines the boolean functions $H_\circ$.

Homomorphisms between two rings R and S extend to homomorphisms between the polynomial rings $R[X]$ and $S[X]$. Similarily, we may extend truth assignments f on Λ to assignments f_* on $\mathrm{SL}(\Lambda)$. This is done recursively. Let f be an arbitrary truth assignment. Define

$$f_*(\neg s) = H_\neg(f_*(s)) \qquad f_*(s \circ w) = H_\circ(f_*(s), f_*(w))$$

then f_* is defined on all of $\mathrm{SL}(\Lambda)$.

Let $s \in \mathrm{SL}(\Lambda)$ be a statement. s is a **tautology**, if, for any truth assignment f on Λ, $f_*(s) = \top$. A **contadiction** conversely satisfies $f_*(s) = \bot$ for all assignments f. If a statement is neither a tautology nor a contradiction, we say it is **contingent**. We summarize the statement "s is a tautology" by $\models s$. We say a statement s **semantically implies** w, written $s \models w$, if $\models s \Rightarrow w$, or correspondingly, if $f_*(w) = \top$ whenever $f_*(s) = \top$.

Suppose that we wish to verify whether $s \in \mathrm{SL}(\Lambda)$ is a tautology. Let $x_1, \dots, x_n \in \Lambda$ be all the variables which occur in s. Define a boolean function $g : \{0,1\}^n \to \{0,1\}$,

$$g(y_1 \dots, y_n) = f_*^{(y_1,\dots,y_n)}(s)$$

where $f^{(y_1,\dots,y_n)}$ is a truth assignment formed by mapping x_i to y_i. This is well defined, because if h and k are two truth assignments which agree on the x_i, then they agree at s. If f is an arbitrary truth assignment, then

$$g(f_*(x_1), \dots, f_*(x_n)) = f_*(s)$$

which can be seen from an easy structural induction. Thus s is a tautology if and only if

$$g(y_1, \dots, y_n) = \top$$

for all choice of y_i. Therefore one need only construct the truth table of g to confirm whether s is a tautology or not. To prevent errors, it is best to construct a truth table containing all subformulas of s, so that one can verify that calculations are consistant with other calculations. This may be done side by side, in the same table.

Example. *For instance, for any variable $A \in \Lambda$, $A \vee \neg A$ is a tautology, $A \wedge \neg A$ is a contradiction, and $\neg A$ is contingent, which is verified by the truth table*

A	$A \vee \neg A$	$A \wedge \neg A$	$\neg A$
$\top$	$\top$	$\bot$	$\bot$
$\bot$	$\top$	$\bot$	$\top$

The first formula is the **law of excluded middle**.

Example. *Let $A \models B$ be a tautology, and suppose that A and B have no variables in common. Then A is a contradiction, or B is a tautology, for if there is a truth assignment on the variables of A which make the statement true, and a truth assignment on B which cause the statement to be false, then we may combine the truth assignments to create a truth assignment in which $A \Rightarrow B$ is false.*

Theorem 2.5 (Semantic Modus Ponens)**.** *If* $\models s$ *and* $s \models w$*, then* $\models w$*.*

Proof. Let f be a truth assignment. Then $f_*(s) = \top$ and

$$f_*(s \Rightarrow w) = H_{\Rightarrow}(f_*(s), f_*(w)) = H_{\Rightarrow}(\top, f_*(w)) = \top$$

This holds only when $f_*(w) = \top$. $\square$

The next theorem relies on a useful string manipulation technique which shall later prove a useful formalism. If $s \in \Lambda^*$, $x = (x_1, \dots, x_n)$ are distinct letters of Λ, and $w = (w_1, \dots, w_n) \in \Lambda^*$, then we shall let

$$s[w_1/x_1, \dots, w_n/x_n] = s[w/x]$$

be the **substitution** of s, denoting the string in Λ^* obtained from swapping the x_i with w_i. It is customary to focus only on alphabets which are countable, but arguments precede in much the same way, and restricting ourselves only makes proofs more complicated, so we do not follow this custom except where it becomes necessary.

Theorem 2.6. *If* $\models w$*, then* $\models w[s/x]$

Proof. Consider a truth assignment f. We shall define another truth assignment $\tilde{f}$ such that $f(v[s/x]) = \tilde{f}(v)$ for all v. Define $\tilde{f}(x_i) = f(s_i)$, and if $y \notin x$, define $\tilde{f}(y) = f(y)$. Our base case, where v is a variable, satisfies the claim by construction. Then, by induction, if $v = (u \circ w)$, then

$$\tilde{f}_*(v) = H_\circ(\tilde{f}_*(u), \tilde{f}_*(w)) = H_\circ(f_*(u[s/x]), f_*(w[s/x])) = f_*(v[s/x])$$

A similar proof answers the case where $u = (\neg v)$. Since w is a tautology,

$$f_*(w[s/x]) = \tilde{f}_*(w) = \top$$

So $w[s/x]$ is a tautology. $\square$

Corollary 2.7. *If* $v \models w$*, then* $v[s/x] \models w[s/x]$*.*

2.3 Truth Functional Completeness

We hope that propositional logic can model all notions of truth, such that all truth functions can be formed from our original set. Here we argue why our logic can model all such notions. Let Λ be a set of boolean functions. The **clone** of Λ is the smallest set containing Λ and all projections $\pi_k^n : \{0,1\}^n \to \{0,1\}$, defined

$$\pi_k^n(x_1, \dots, x_n) = x_k$$

and in addition, if $g : \{0,1\}^n \to \{0,1\}$ and $f_1, \dots, f_n : \{0,1\}^m \to \{0,1\}$ are in the clone, then so is $g(f_1, \dots, f_n) : \{0,1\}^m \to \{0,1\}$. Λ is **truth functionally complete** if its clone is the set of all boolean functions.

Example. *$\{H_\neg, H_\wedge, H_\vee\}$ is a truth functionally complete, since every formula can be put in* **conjunctive normal form***, for we may write the 'true and false' functions*

$$\top(x) = \top = H_\vee(H_\neg(x), x) \qquad \bot(x) = \bot = H_\wedge(H_\neg(x), x)$$

and then we write

$$f(x_1, \dots, x_n) = \bigvee_{\substack{(y_1,\dots,y_n) \in \{0,1\}^n \\ f(y_1,\dots,y_n) = \top}} \bigwedge_{i=1}^n H_\Leftrightarrow(x_i, y_i)$$

where we identify an element of $\{\top, \bot\}$ with the constant boolean function, define $\bigcirc_{i=1}^n f_i = H_\circ(f_n, \bigcirc_{i=1}^{n-1})$, and take

$$H_\Leftrightarrow(x,y) = H_\wedge(H_\Rightarrow(x,y), H_\Rightarrow(y,x)) = H_\wedge(H_\vee(H_\neg(x), y), H_\vee(H_\neg(y), x))$$

This can be further reduced, by noticing that

$$H_\wedge(x,y) = H_\neg(H_\vee(H_\neg(x), H_\neg(y)))$$

which is a truth functional form of Boole's inequality, implying $\{\neg, \vee\}$ is truth functionally complete. We can also consider **disjunctive normal form***, left to the reader to define.*

We see that conjunctive and normal disjunctive forms are a way to canonically represent certain truth functions. Say a formula s is **satisfiable**, in the sense that there is some truth assignment f such that $f_*(s) = \top$. It is a standard computational problem to verify whether such a formula is satisfiable, as a great many problems can be reduced to satisfiability. For instance, say one wishes to verify whether a graph (V, E) is m colorable (that is, there is a function $f : V \to \{1, \dots, m\}$ such that if $(v, w) \in E$, $f(v) \neq f(w)$). For each vertex $v \in V$ and color $i \in \{1, \dots, m\}$, let v_i be a variable. A graph is colorable if and only if the statement

$$\bigwedge_{v \in V} \left(\bigvee_{i=1}^{m} v_i \right) \wedge \bigwedge_{(v,w) \in E} \left(\bigwedge_{i=1}^{m} (\neg v_i \vee \neg w_i) \right) \wedge \bigwedge_{v \in V} \left(\bigwedge_{i,j=1}^{m} (v_i \vee \neg v_j) \right)$$

is satisfiable (the first big clause says that some color is assigned to each vertex, the last that the color is unique. The middle clause is the coloring constraint). If a formula is in disjunctive normal form, it is algorithmically easy to verify whether the formula is satisfiable. We just need to check whether one of the disjunctive clauses is consistant, which can be done in a time proportional to the size of the formula. Checking whether a formula is in conjunctive normal form is much more difficult – in fact, in computability theory one discovers that almost all interesting problems can be reduced to a satisfiability problem, so if it was easy to determine if a CNF is solvable, then we could solve a great many problems easily, which sound unlikely. To determine if the problem is easy is the $\mathbf{P} = \mathbf{NP}$ problem, one of the most fundamental problems in computing science.

Example. *The mathematician Henry M. Sheffer found a single truth function which is truth functionally complete. Consider the* **Sheffer stroke** $x|y$*, also known as* **NAND***, defined by the truth table*

x	y	$H_{\mid}(x,y)$
$\bot$	$\bot$	$\top$
$\bot$	$\top$	$\top$
$\top$	$\bot$	$\top$
$\top$	$\top$	$\bot$

Then $H_\neg(x) = H_{\mid}(x,x)$*, and* $H_\vee(x,y) = H_{\mid}(H_\neg(x), H_\neg(y))$*, which implies, since this set is truth functionally complete, that the sheffer stroke is truth functionally complete.*

The previous example is incredibly important to circuit design. Logical statements can be represented by boolean functions. Since all truth functions can be built from the sheffer stroke, we need only make an atomic circuit for the sheffer stroke, and then all other circuits are constructed by combining sheffer strokes. Computers are constructed from millions of NAND gates.

2.4 Deduction

When mathematicians want to derive whether a statement is true, they do not construct truth functions. Instead, they argue *why* the statement is true. Here we shall provide the mechanics for modelling a mathematical argument. We will show that truth tables and arguments are equivalent – a statement is a tautology if and only if it can be proved. This is known as a *completeness result*, for it says that our semantic understanding of a theory is the same as the deductive understanding.

First, we thin out the connectives in our theory. Since $\Rightarrow$ and $\neg$ are truth functionally complete, we can consider a system consisting only of these connectives, and reinterpret other formulas as semantically equivalent formulas in the reduced theorem. Next, we defined the **theorems** of $SL(\Lambda)$, which are elements of the smallest set such that for any formulas s, w, u,

1. Any axiom is a theorem, where the axioms are any statement of the form

$$\begin{aligned}
&(A1) && s \Rightarrow (w \Rightarrow s)\\
&(A2) && (s \Rightarrow (w \Rightarrow u)) \Rightarrow ((s \Rightarrow w) \Rightarrow (s \Rightarrow u))\\
&(A3) && (\neg s \Rightarrow \neg w) \Rightarrow ((\neg s \Rightarrow w) \Rightarrow s)
\end{aligned}$$

2. Modus Ponens holds in our system. If $s \Rightarrow w$ and s are theorems, then w is a theorem. When we apply modus ponens, we may write that the theorem was obtained by (MP).

We shall write $\vdash s$ to state that s is a theorem.

For statements, the 'smallness' characterization gives us structural induction. For theorems, smallness gives us a abstract notion of a 'proof'. A statement s is a theorem of $SL(\Lambda)$ if and only if there is a sequence of formulae $(s_1, \ldots, s_n)$ such that $s_n = s$, and each s_i is either an axiom, or is obtained from some s_j and s_k by modus ponens, where $j, k < i$. This sequence is known as a proof. Often, we list a proof from top to bottom, where we reference how we obtained each element of the sequence alongside the proof.

Example. *Let us construct a proof of $\vdash s \Rightarrow s$, for any $s \in SL(\Lambda)$.*

$\mathsf{s} \Rightarrow \mathsf{s}$	
1. $(\mathsf{s} \Rightarrow ((\mathsf{s} \Rightarrow \mathsf{s}) \Rightarrow \mathsf{s}))$	*(A1)*
2. $(\mathsf{s} \Rightarrow ((\mathsf{s} \Rightarrow \mathsf{s}) \Rightarrow \mathsf{s})) \Rightarrow ((\mathsf{s} \Rightarrow (\mathsf{s} \Rightarrow \mathsf{s})) \Rightarrow (\mathsf{s} \Rightarrow \mathsf{s}))$	*(A2)*
3. $((\mathsf{s} \Rightarrow (\mathsf{s} \Rightarrow \mathsf{s})) \Rightarrow (\mathsf{s} \Rightarrow \mathsf{s}))$	*(1),(2),(MP)*
4. $(\mathsf{s} \Rightarrow (\mathsf{s} \Rightarrow \mathsf{s}))$	*(A1)*
5. $\mathsf{s} \Rightarrow \mathsf{s}$	*(3),(4),(MP)*

In future proofs, we shall be able to use $\vdash s \Rightarrow s$ implicitly, since we now know the statement can be proved in any of its forms. We will denote its application by (I).

In mathematics, we work in systems where implicit assumptions are made. In group theory, we assume that operations are associative. In geometry, we assume there is a line between any two points. To perform mathematics, we add additional axioms to logic, and prove results from these axioms. If Γ is a subset of $\mathrm{SL}(\Lambda)$, then we may consider each member of Γ to be an axiom. We write $\Gamma \vdash s$ if one may prove s assuming all formulae in Γ have already been proved. That is, we may write a sequence $(s_1, \ldots, s_n)$, where $s_n = s$, and each s_i is either an axiom, an element of Γ, or is obtained by modus ponens from previous elements of the sequence.

Theorem 2.8 (Deduction Theorem)**.** *If $\Gamma \cup \{s\} \vdash w$, then $\Gamma \vdash s \Rightarrow w$.*

Proof. We prove the theorem by induction of the size of the proof of w. Consider a particular proof $(s_1, \ldots, s_n)$ of w from $\Gamma \cup \{s\}$. Suppose that $n = 1$. Then $s_1 = w$, and w must either be an axiom, an element of Γ, or equal to s. In the first and second case, the proof is equally valid in Γ, and so $\Gamma \vdash s \Rightarrow w$ follows from the axiom $(w \Rightarrow (s \Rightarrow w))$. If $s = w$, Then we have shown that $\vdash w \Rightarrow w$, so obviously $\Gamma \vdash s \Rightarrow w$.

Now we consider the problem proved for $m < n$. $s_n = w$ is either an axiom, an element of Γ, equal to s, or proved by modus ponens from $s_i =$

$(u \Rightarrow w)$ and $s_j = u$. We have already shown all but the last case. By induction, $\Gamma \vdash s \Rightarrow (u \Rightarrow w)$ and $\Gamma \vdash s \Rightarrow u$. But

$$(s \Rightarrow (u \Rightarrow w)) \Rightarrow ((s \Rightarrow u) \Rightarrow (s \Rightarrow w))$$

is an axiom, so $\Gamma \vdash s \Rightarrow w$. □

Example. *For any statements s and w,* $\{s \Rightarrow w, w \Rightarrow u\} \vdash s \Rightarrow u$. *This follows from a basic application of (A2). But this implies the two cut rules, that*

$$\vdash (s \Rightarrow w) \Rightarrow ((w \Rightarrow u) \Rightarrow (s \Rightarrow u))$$
$$\vdash (w \Rightarrow u) \Rightarrow ((s \Rightarrow w) \Rightarrow (s \Rightarrow u))$$

which are much more tricky to prove (though technically a proof may be constructed inductively from the proof of the deduction theorem).

Example. *Let us prove the double negation elimination axiom,* $\vdash \neg\neg s \Rightarrow s$ *by the deduction theorem.*

$\neg\neg s \Rightarrow s$	
1. $\neg\neg s$	
2. $(\neg s \Rightarrow \neg\neg s) \Rightarrow ((\neg s \Rightarrow \neg s) \Rightarrow s)$	*(A3)*
3. $(\neg\neg s) \Rightarrow (\neg s \Rightarrow \neg\neg s)$	*(A1)*
4. $(\neg s \Rightarrow \neg\neg s)$	*(1),(3),(MP)*
5. $(\neg s \Rightarrow \neg s) \Rightarrow s$	*(2),(4),(MP)*
6. $\neg s \Rightarrow \neg s$	*(I)*
7. s	*(5),(6),(MP)*
8. $\neg\neg s \Rightarrow s$	*(1-7),(DT)*

In future proofs, application of the statement will be denoted $(\neg\neg E)$. *Now lets prove negation introduction,* $\vdash s \Rightarrow \neg\neg s$.

$s \Rightarrow \neg\neg s$	
1. $\neg\neg\neg s \Rightarrow \neg s$	$(\neg\neg E)$
2. s	
3. $(\neg\neg\neg s \Rightarrow \neg s) \Rightarrow ((\neg\neg\neg s \Rightarrow s) \Rightarrow \neg\neg s)$	*(A3)*
4. $(\neg\neg\neg s \Rightarrow s) \Rightarrow \neg\neg s$	*(1), (3), (MP)*
5. $s \Rightarrow (\neg\neg\neg s \Rightarrow s)$	*(A1)*
6. $\neg\neg\neg s \Rightarrow s$	*(2),(5),(MP)*
7. $\neg\neg s$	*(4),(6),(MP)*
8. $s \Rightarrow \neg\neg s$	*(2-7), (DT)*

We shall denote this rule $(\neg\neg I)$.

Example. *Lets prove* $\neg w \vdash w \Rightarrow u$*, by proving* $\neg w, w \vdash u$.

$$
\begin{array}{lll}
 & \neg w \Rightarrow (w \Rightarrow u) & \\
\hline
 & 1.\ \neg w & \\
 & 2.\ w & \\
 & 3.\ (\neg u \Rightarrow \neg w) \Rightarrow ((\neg u \Rightarrow w) \Rightarrow u) & (A3) \\
 & 4.\ \neg w \Rightarrow (\neg u \Rightarrow \neg w) & (A1) \\
 & 5.\ \neg u \Rightarrow \neg w & (1),(4),\ (MP) \\
 & 6.\ (\neg u \Rightarrow w) \Rightarrow u & (3),(5),(MP) \\
 & 7.\ w \Rightarrow (\neg u \Rightarrow w) & (A1) \\
 & 8.\ \neg u \Rightarrow w & (2),(7),(MP) \\
 & 9.\ u & (6),(8),(MP) \\
 & 10.\ w \Rightarrow u & (2\text{-}9),(DT) \\
 & 11.\ \neg w \Rightarrow (w \Rightarrow u) & (1\text{-}10),\ (DT)
\end{array}
$$

This is a proof of **the law of contradiction***, denoted* (LC).

Example. *Consider the following proof.*

$$
\begin{array}{lll}
 & (s \Rightarrow w) \Rightarrow (\neg w \Rightarrow \neg s) & \\
\hline
 & 1.\ s \Rightarrow w & \\
 & 2.\ \neg w & \\
 & 3.\ \neg w \Rightarrow (\neg\neg s \Rightarrow \neg w) & (A1) \\
 & 4.\ \neg\neg s \Rightarrow \neg w & (2),(3),(MP) \\
 & 5.\ (\neg\neg s \Rightarrow \neg w) \Rightarrow ((\neg\neg s \Rightarrow w) \Rightarrow \neg s) & (A3) \\
 & 6.\ (\neg\neg s \Rightarrow w) \Rightarrow \neg s & (4),(5),(MP) \\
 & 7.\ \neg\neg s & \\
 & 8.\ \neg\neg s \Rightarrow s & (\neg\neg E) \\
 & 9.\ s & (7),(8),(MP) \\
 & 10.\ w & (1),(9),(MP) \\
 & 11.\ \neg\neg s \Rightarrow w & (7\text{-}10),\ (DT) \\
 & 12.\ \neg s & (6),(11),\ (MP) \\
 & 13.\ \neg w \Rightarrow \neg s & (2),(12),(MP) \\
 & 11.\ (s \Rightarrow w) \Rightarrow (\neg w \Rightarrow \neg s) & (1\text{-}10),\ (DT)
\end{array}
$$

This is the **law of contraposition** (LCP).

Example. *Lets prove* $\vdash (s \Rightarrow w) \Rightarrow ((\neg s \Rightarrow w) \Rightarrow w)$.

$$\begin{array}{lll}
& (\mathsf{s} \Rightarrow \mathsf{w}) \Rightarrow ((\neg \mathsf{s} \Rightarrow \mathsf{w}) \Rightarrow \mathsf{w}) & \\
& \quad 1.\ \mathsf{s} \Rightarrow \mathsf{w} & \\
& \quad 2.\ (\mathsf{s} \Rightarrow \mathsf{w}) \Rightarrow (\neg \mathsf{w} \Rightarrow \neg \mathsf{s}) & (LCP) \\
& \quad 3.\ \neg \mathsf{w} \Rightarrow \neg \mathsf{s} & (1),(2),(MP) \\
& \qquad 4.\ \neg \mathsf{s} \Rightarrow \mathsf{w} & \\
& \qquad 5.\ (\neg \mathsf{w} \Rightarrow \neg\neg \mathsf{s}) & (4),(LCP) \\
& \qquad 6.\ \neg\neg \mathsf{s} \Rightarrow \mathsf{s} & (\neg\neg E) \\
& \qquad 7.\ (\neg \mathsf{w} \Rightarrow \neg\neg \mathsf{s}) \Rightarrow ((\neg\neg \mathsf{s} \Rightarrow \mathsf{s}) \Rightarrow (\neg \mathsf{w} \Rightarrow \mathsf{s})) & (CUT) \\
& \qquad 8.\ (\neg\neg \mathsf{s} \Rightarrow \mathsf{s}) \Rightarrow (\neg \mathsf{w} \Rightarrow \mathsf{s}) & (5),(7),(MP) \\
& \qquad 9.\ \neg \mathsf{w} \Rightarrow \mathsf{s} & (6),(8),(MP) \\
& \qquad 10.\ (\neg \mathsf{w} \Rightarrow \neg \mathsf{s}) \Rightarrow ((\neg \mathsf{w} \Rightarrow \mathsf{s}) \Rightarrow \mathsf{w}) & (A3) \\
& \qquad 11.\ (\neg \mathsf{w} \Rightarrow \mathsf{s}) \Rightarrow \mathsf{w} & (3),(10),(MP) \\
& \qquad 12.\ \mathsf{w} & (9),(11),(MP) \\
& \quad 13.\ (\neg \mathsf{s} \Rightarrow \mathsf{w}) \Rightarrow \mathsf{w} & (2-9),(DT) \\
& 14.\ (\mathsf{s} \Rightarrow \mathsf{w}) \Rightarrow ((\neg \mathsf{s} \Rightarrow \mathsf{w}) \Rightarrow \mathsf{w}) & (1-10),(DT)
\end{array}$$

This essentially proves that $(s \vee \neg s) \Rightarrow w$ *implies* w.

We shall verify that proofs never lead to contradiction.

Theorem 2.9. *If* $\vdash s$*, then* $\vDash s$.

Proof. This is a trivial proof by structural induction. Prove that all axioms are tautologies, and that the set of tautologies is closed under modus ponens. □

This theorem shows that there are some statements which are not provable in our system. In fact, if s is provable, then $\neg s$ is not provable. We call an axiom system like this **absolutely consistant**. We shall show that all tautologies are provable, which shows the system is **complete**. A complete system is effectively one in which all theorems which were meant to be able to be proved, are able to be proved.

Lemma 2.10. *Let* $x_1, \dots, x_n \in \Lambda$ *be variables in* $s \in SL(\Lambda)$*. Let* f *be a truth assignment, and define* $s' = s$ *if* $f_*(s) = \top$*, or* $s' = \neg s$ *if* $f_*(s) = \bot$*. Then*

$$x'_1, \dots, x'_n \vdash s'$$

Proof. We prove by structural induction. If $s = x_1$, then $x_1' = s'$, and $s \vdash s$ is a trivial theorem. If $s = \neg w$, we consider two cases. If $w' = w$, then $s' = \neg\neg w$, and we have already shown $w \vdash \neg\neg w$, hence $x_1', \dots, x_n' \vdash s'$. If $w' = \neg w$, then $s' = w'$, and the theorem is trivial. If $s = w \Rightarrow u$, then either $f_*(w) = \top$ and $f_*(u) = \top$, or $f_*(w) = \bot$. In the first case, we have $x_1', \dots, x_n' \vdash u$, from which $x_1', \dots, x_n' \vdash w \Rightarrow u$ follows. In the second case, $x_1', \dots, x_n' \vdash \neg w$, and we have $\neg w \vdash (w \Rightarrow u)$. □

Corollary 2.11 (Completeness Theorem). *If* $\models s$, $\vdash s$.

Proof. Let $x_1, \dots, x_n$ be the variables in s. By the last lemma, we have

$$x_1, \dots, x_n \vdash s \qquad x_1, \dots, \neg\neg x_n \vdash s$$

By the deduction theorem

$$x_1, \dots, x_{n-1} \vdash x_n \Rightarrow s \qquad x_1, \dots, x_{n-1} \vdash \neg x_n \Rightarrow s$$

But then, since $\vdash (x_n \Rightarrow s) \Rightarrow ((\neg x_n \Rightarrow s) \Rightarrow s)$,

$$x_1, \dots, x_{n-1} \vdash s$$

By induction, we find $\vdash s$. □

Notice that every proof we have given leading to the completeness theorem is constructive – that is, one could effectively write an algorithm which constructs the items in the proof. This means that propositional logic is *decidable*, there is an effective algorithm that takes a statement of propositional logic, and returns a proof of that statement, if such a proof exists. The completeness theorem also shows that the *decision* problem of propositional logic is also solvable. Given a statement of propositional logic, to determine whether we can prove the statement, we need only test whether the statement is true under all truth interpretations.

Before we finish our discussion of semantics, we note that there are many other axioms systems which can be used to define a propositional calculus (in the sense that they prove all tautologies). Most interesting is the axiom system whose only connective is the sheffer stroke, and whose only axiom schema is

$$(B|(C|D))|[\{E|(E|E)\}|[(F|C)|((B|F)|(B|F))]]$$

and whose rule of inference is to infer D from $B|(C|D)$, and B. Of course, it is outlandish to attempt proofs in such a system, hence why we did not attempt this chapter using the system.

2.5 Verifying Propositional Formulas

Because of the completeness theorem, verifying that a particular formula of propositional logic is provable is reduced to an algorithm. If a formula s contains variables $x_1, \dots, x_n$, we simply have to consider all truth assignments $f : \{x_1, \dots, x_n\} \to \{\top, \bot\}$, and then check that $f_*(s) = \top$. The completeness theorem tells us a formula is provable if and only if it is true under all truth assignments, so this suffices to decide whether the formula is provable. For any formula s of length n, there can be at most n variables in s, so the algorithm runs in time bounded by $O(n2^n)$. This is a tight bound, because there are formulas s of arbitrarily large length n containing $O(n)$ variables. Clearly, such a runtime is undesirable, but without assuming the famous $\mathbf{P} = \mathbf{NP}$ conjecture, it is doubtful whether we can do much better. Here we introduce some techniques, which work in particular circumstances, and simplify the verification of some tautology.

Theorem 2.12 (Craig Interpolation)**.** *Let $A \models B$, and let the set of variables shared by A and B be $x_1, \dots, x_n$. Then there is a statement C, known as the* **interpolant**, *containing only the variables x_i, such that $A \models C$ and $C \models B$.*

Proof. We proceed by induction on the number of variables in A which do not occur in B. If every variable in A occurs in B, let $C = A$. In the general case, fix some variable x in A but not in B, take y in both A and B, and define

$$D = A[(y \wedge \neg y)/x] \vee A[(y \vee \neg y)/x]$$

If $f_*(A) = \top$, and $f_*(x) = \bot$, then $f_*(A[(y \wedge \neg y)/x]) = \top$. If $f_*(x) = \top$, then $f_*(A[(y \vee \neg y)/x]) = \top$, so $A \models D$. In addition, $D \models B$. Let $f_*(D) = \top$. Then $f_*(A[(y \wedge \neg y)/x]) = \top$, or $f_*(A[(y \vee \neg y)/x]) = \top$. In the first case, we modify the truth assignment f so that $f_*(x) = \bot$ (without changing the values of B or D). Then $f_*(A) = \top$, so $f_*(B) = \top$. The other case follows by letting $f_*(x) = \top$. By induction, there is an interpolant C for which $D \models C$ and $C \models B$, and then $A \models C$, since $(x \Rightarrow y) \wedge (y \Rightarrow z) \models x \Rightarrow z$. $\square$

Now suppose we wish to verify a formula of the form $s \Rightarrow w$ is satisfiable, where the number of variables of s and w is few in number. The above theorem provides a constructive way to find a formula u containing only the variables that occur in both s and w, such that $u \Rightarrow w$ holds if and only if $s \Rightarrow w$ holds, and this vastly simplifies the truth table calculation. However, if s is length n with a variables, w is length m with b variables, and s and w share c variables in common, then the u constructed above will have length $O(n2^{a-c})$, and the calculation of the truth table of $u \Rightarrow w$ will take $O((n2^{a-c} + m)2^b)$ time, which is still exponential, but in certain cases can be feasible to calculate. This is especially true if we can simplify u to a simpler form by eliminating redundant parts of the logic.

Example. *Consider $A = (y_1 \Rightarrow x) \wedge (x \Rightarrow y_2)$ and $B = (y_1 \wedge z) \Rightarrow (y_2 \wedge z)$. Then $A \vDash B$. We shall find an interpolant.*

$$\begin{aligned} C = &[(y_1 \Rightarrow (y_1 \vee \neg y_1)) \wedge ((y_1 \vee \neg y_1) \Rightarrow y_2)] \\ &\vee [(y_1 \Rightarrow (y_1 \wedge \neg y_1)) \wedge ((y_1 \wedge \neg y_1) \Rightarrow y_2)] \end{aligned}$$

This equation can be simplified to $\neg y_1 \vee y_2$.

The second technique to reduce the complexity of verifying an equation is known as **resolution**. We assume the formula we are given is in conjunctive normal form. Let $w = s_1 \wedge \cdots \wedge s_n$ be be such a formula of propositional logic. We will view a conjunction of clauses as a set $w = \{s_1, \dots, s_n\}$, where w is true in an interpretation only when each s_i is true. Without loss of generality, we may assume that no s_i contains A and $\neg A$ in disjunctions, for then the conjunct is vacously satisfied. Suppose s_i contains an instance of a statement A, and s_j contains an instance of a statement $\neg A$. The disjunction obtained from s_i and s_j by concatenating all disjuncts together, dropping all occurrences of A and $\neg A$, as well as removing duplicates, is called the resolution of s_i and s_j with respect to the variable A, and we will denote the resulting formula by $\text{res}_A(s_i, s_j)$. Then $s_i, s_j \vDash \text{res}_A(s_i, s_j)$, because semantically, either A is true or $\neg A$ is true, and in the first case some clause of s_j other than $\neg A$ must be true, and in the second some clause of s_i other than A must be true. Given w, let $\text{res}(w)$ be the smallest set of clauses such that $w \subset \text{res}(w)$, and if s_i and s_j are in $\text{res}(w)$, where s_i contains some A and s_j contains some $\neg A$, then $\text{res}_A(s_i, s_j) \in \text{res}(w)$. It is clear that w is true if and only if $\text{res}(w)$ is true, because we have only added clauses which are logically implied by the other clauses of w. This is the **resolution** of w.

Example. *If*

$$w = \{A \vee \neg B \vee \neg C, \neg A \vee \neg B \vee D, A \vee C \vee D\} = \{\alpha, \beta, \kappa\}$$

then $res(w)$ contains the clauses of w, and in addition the clauses

$$res_A(\alpha, \beta) = \neg B \vee \neg C \vee D$$

$$res_A(\beta, \kappa) = \neg B \vee C \vee D$$
$$res_C(\alpha, \kappa) = A \vee \neg B \vee D$$
$$res_C(res_A(\alpha, \beta), res_A(\beta, \kappa)) = \neg B \vee D$$

If $\mathrm{res}_A(s, s')$ ever returns the empty clause for some $s, s' \in \mathrm{res}(w)$, then $s = A$, $s' = \neg A$, and we conclude that $\mathrm{res}(w)$ is a contradiction, hence w is a contradiction. The converse is true, though it's a little tricky to prove.

Lemma 2.13. *Let $res_C(w)$ denote the term obtained by concatenating all possible terms available from the s_i by resolution on the variable C, and then removing all clauses which contain an instance of C or $\neg C$. Then if $res_C(w)$ is satisfiable, and contains no empty clauses, then w is satisfiable.*

Proof. Let f be a truth assignment with $f_*(\mathrm{res}_C(w)) = \top$. We then claim that by setting $f(C) = \top$ or $f(C) = \bot$, we find that $f_*(w) = \top$. If s_i is a clause of w not containing C or $\neg C$, then $f_*(s_i) = \top$. Conversely, if no s_i contains $\neg C$, then we may set $f(C) = \top$, and then all clauses are satisfied. Similarily, if no s_i contains C, we set $f(C) = \bot$. Thus we are reduced to the case where some clause of w contains C, and some other clause contains $\neg C$. Let

$$g(A) = \begin{cases} \top & A = C \\ f(A) & A \neq C \end{cases} \qquad h(A) = \begin{cases} \bot & A = C \\ f(A) & A \neq C \end{cases}$$

Suppose that $g_*(s_i) = \bot$, and $h_*(s_j) = \bot$. Then s_i must contain an instance of C, and s_j must contain an instance of $\neg C$. We note that $s' = \mathrm{res}_C(s_i, s_j)$ contains no instances of C or $\neg C$ and is nonempty, hence $f_*(s') = g_*(s') = h_*(s') = \top$, hence there is some clause in s_i or s_j already satisfied by the interpretation of s. This implies that the condition $g_*(s_i) = \bot$ and $h_*(s_j) = \bot$ is impossible, so either $g_*(s_i) = \top$ for all i, or $h_*(s_j) = \top$ for all j, and this completes the proof. ☐

Theorem 2.14. *w is a contradiction if and only if $res(w)$ contains an empty clause.*

Proof. We prove by induction on the number of atoms in w. If w contains only a single atom A, then either w is satisfiable, and $w = \{A\}$ or $w = \{\neg A\}$, or w is not satisfiable and $w = \{A, \neg A\}$, hence the theorem is satisfied. Now given any w, either $\mathrm{res}_A(w)$ contains an empty clause, or $\mathrm{res}_A(w)$ must be a contradiction. In the second case, we apply induction on the number of atoms to conclude that $\mathrm{res}(\mathrm{res}_A(w))$ contains an empty clause, and $\mathrm{res}(\mathrm{res}_A(w)) \subset \mathrm{res}(w)$. ☐

Resolution gives us an algorithm to calculate whether any formula of prepositional logic is true. Given such a statement s, take $\neg s$, and convert it to a conjunctive normal form w. Then s is a tautology if and only if w is a contradiction, so we just determine if $\text{res}(w)$ contains an empty clause, and this tells us if s is a tautology.

2.6 Sequent Calculi

The complete formal system we have studied is styled in the sense of a great many formal systems, known as **Hilbert systems**. A Hilbert system just takes axioms, and deductive rules, and then forms proofs as sequences $(s_1, \dots, s_n)$. But there are a great many styles of formal systems, and this section I shall detail my personal favourite, natural deduction. Most actual proofs in mathematics do not follow a linear style. We instead form a proof by combining prior deductions in a non-linear way to reach the conclusion, the end of the proof. Thus natural deduction does not model a proof as a sequence $(s_1, \dots, s_n)$, but instead as a tree, whose root node is the conclusion we are attempting to form. The nodes of the tree will not consist of formulas, but instead of sequents, which we have almost already seen, which are pairs of sequences of formulas of the form $s_1, s_2, \dots, s_n \vdash w_1, \dots, w_m$, which we interpret as proving $(s_1 \wedge s_2 \wedge \dots \wedge s_n) \Rightarrow (w_1 \vee w_2 \vee \dots \vee w_m)$. Why the assymmetry? It turns out that this will give us symmetry in proofs, which we shall require later. A manifestation of this symmetry is that if $s \Rightarrow w$ is true, then both $(s \wedge u) \Rightarrow w$ and $s \Rightarrow (w \vee u)$ is true, so the sequents $s, u \vdash w \quad s \vdash u, w$ may be derived from the sequent $s \vdash w$. We have already treated the semantics of propositional logic, so we may just state the axioms and deduction rules. Unlike a hilbert system, our system has far more deduction rules than axioms, which is why this system is more *natural* – we more naturally deal with deduction rules. The only axioms are of the form

$$s \vdash s$$

the deduction rules are the edges from which we form our tree,

$$\frac{\Gamma, s \vdash \Delta}{\Gamma, s \wedge w \vdash \Delta}\ (\wedge L) \qquad \frac{\Gamma \vdash s, \Delta}{\Gamma \vdash w \vee s, \Delta}\ (\vee R)$$

$$\frac{\Gamma, s \vdash \Delta \qquad \Pi, w \vdash \Delta}{\Gamma, \Pi, s \vee w \vdash \Delta}\ (\vee L) \qquad \frac{\Gamma \vdash s, \Delta \qquad \Gamma \vdash w, \Pi}{\Gamma \vdash s \wedge w, \Delta, \Pi}\ (\wedge R)$$

$$\frac{\Gamma \vdash s, \Delta \qquad \Sigma, w \vdash \Pi}{\Gamma, \Sigma, s \Rightarrow w \vdash \Delta, \Pi}\ (\Rightarrow L) \qquad\qquad \frac{\Gamma, s \vdash w, \Delta}{\Gamma \vdash s \Rightarrow w, \Delta}\ (\Rightarrow R)$$

$$\frac{\Gamma \vdash s, \Delta}{\Gamma, \neg s \vdash \Delta}\ (\neg L) \qquad\qquad \frac{\Gamma, s \vdash \Delta}{\Gamma \vdash \neg s, \Delta}\ (\neg R)$$

We have additional deduction rules, known as structural rules, which help show that sequents are the same, without any real logical content.

$$\frac{\Gamma \vdash \Delta}{\Gamma' \vdash \Delta'}\ (\mathrm{P})$$

in the last rule, we mean that Γ' is obtained from Γ by permuting the sequence, the same for Δ', as well as removing or adding duplicates. Thus a proof of a sequent $\Gamma \vdash \Delta$ is a tree, whose root is $\Gamma \vdash \Delta$, whose leaves are axioms of the form $s \vdash s$, and such that each edge is annotated by the appropriate deduction rule, for which the deduction is accurate. It turns out it is fairly simple to form deductions, since we may work backwards in most cases to determine which edges to apply.

Example. *Consider a proof of Pierce's law, $(s \Rightarrow w) \Rightarrow s) \vdash s$. To prove this, we likely need to apply $(\Rightarrow L)$, so we must prove $\vdash (s \Rightarrow w), s$ and $s \vdash s$. Since sequent calculus is meant to model propositional logic, and we will soon prove this system complete, we know we can prove $\vdash (s \Rightarrow w), s$, and this is obtained from $(\Rightarrow R)$ from the sequent $s \vdash w, s$, and this is obtained from (P) from the axiom $s \vdash s$. We obtain the following proof tree.*

$$\frac{\dfrac{\dfrac{s \vdash s}{s \vdash w, s}\ (P)}{\vdash (s \Rightarrow w), s}\ (\Rightarrow I) \qquad s \Rightarrow s}{(s \Rightarrow w) \Rightarrow s) \vdash s}\ (\Rightarrow L)$$

Thus this is a theorem of propositional logic.

We shall require some other sequents to be provable, but we also desire a simple system, one which invokes the cut rule

$$\frac{\Gamma \vdash \Delta, s \qquad s, \Sigma \vdash \Pi}{\Gamma, \Sigma \vdash \Delta, \Pi}\ (\mathrm{CUT})$$

But this is distinctly different from the other rules, for it removes complexity from formulas rather than adds complexity. Nonetheless, we shall prove that we do not need the cut rule – it can always be removed from proofs (It cannot be proved in the Sequent calculi, just removed provided we start from the basic axioms). We must identify properties of rules which can be removed in above applications of the formulae.

Define the **degree** of a formula s to be the height of the parse tree of s, define the degree of a sequent to be the sum of the degrees of the nodes in the sequent which are not the root node (so the degree of a variable is 0). The degree of a cut is the sum of the degrees of the two sequents which form the premise of the deduction rule.

which applies when Σ and Δ contain some common formula s, where Σ^* and Δ^* both have this formula removed. The mix rule is a generalization of the cut rule, so obvious the cut rule can be proved in the mix rule. Conversely, the cut rule implies the mix rule, by a simple induction. The proof is nasty, and can be skipped –

Theorem 2.15. *If a sequent $\Gamma \vdash \Delta$ is provable using the cut rule, then it is provable without the use of the cut rule.*

Proof. The contraction measure of the cut is the number of variables removed by a (P) rule before the cut, and the rank of the cut is the...

We perform a triple induction, first on the rank, then on the contraction measure, and then on the degree. It is clear the rank of a cut is at least 2. The base case occurs when the degree and contraction measure is zero. This implies the use of the cut rule occurs right after introduction of axioms, and the cut rule is of the form

$$\frac{s \vdash s \qquad s \vdash s}{s \vdash s} \text{ (CUT)}$$

clearly, such a derivation is redundant.

If the degree of the cut is zero, then we apply a cut rule consisting only of propositional variables

$$\frac{w \vdash u, s \qquad s, t \vdash v}{w, t \vdash u, v} \text{ (CUT)}$$

□

We first note that the cut rule is *not provable* using the axioms of the calculus, but just that it may always be replaced by a more complicated proof.

Example. *The sequent $s, s \Rightarrow w \vdash w$ is provable*

$$\frac{s \vdash s \qquad w \vdash w}{s, s \Rightarrow w \vdash w} \ (\Rightarrow L)$$

The sequent $\Rightarrow L$ is essentially modus ponens.

We desire to show this system is complete. To do this, we could cheat. Since we already have completeness of a Hilbert system, we just need to form an encoding of the Hilbert system in this system, and an encoding of the sequent calculus in the Hilbert system, such that all axioms are provable. Consider the translation f of the sequent calculus into the Hilbert system, by the map

$$f(\text{“}s_1, \dots, s_n \vdash w_1, \dots, w_m\text{”}) = (s_1 \wedge \dots \wedge s_n) \Rightarrow (w_1 \vee \dots \vee w_m)$$

and the translation g of the Hilbert system into the sequent calculus, defined by

$$g(s) = \text{“}\vdash s\text{”}$$

We define a sequent $s_1, \dots, s_n \vdash w_1, \dots w_m$ to be semantically true, if, under every truth assignment that makes each s_i true, one of the w_j is true. Proving the system is sound is fairly trivial. First, we prove that f and g preserve provability. If S is provable in the sequent calculus, then $f(x)$ is provable in the Hilbert system, and if s is provable in the Hilbert system, then $g(y)$ is provable in the Sequent calculus (we need only prove the axioms and deductions). Second, we prove that f and g preserve semantic completeness, that if S is a semantically true sequent, then so if $f(S)$, and if $\vDash s$, then $g(s)$ is semantically true. This allows us to prove completeness, suppose that $s_1, \dots, s_n \vdash w_1, \dots, w_m$ is semantically true. Then

$$\vDash (s_1 \wedge \dots \wedge s_n) \Rightarrow (w_1 \vee \dots \vee w_m)$$

By completeness of the Hilbert system,

$$\vdash (s_1 \wedge \dots \wedge s_n) \Rightarrow (w_1 \vee \dots \vee w_m)$$

Hence the sequent

$$\vdash (s_1 \wedge \dots \wedge s_n) \Rightarrow (w_1 \vee \dots \vee w_m)$$

is provable in the sequent calculus (note the last formula is completely different from this formula, it is unfortunate the equations coincide). We require that we have already shown that

$$(s \Rightarrow w), s \vdash w$$

are provable sequents. This may be combined with the previous sequent, letting $s = (s_1 \wedge \cdots \wedge s_n)$, $w = (w_1 \vee \cdots \vee w_m)$, and applying the cut rule, we find

$$(s_1 \wedge \cdots \wedge s_n) \vdash (w_1 \vee \cdots \vee w_m)$$

TODO: Finish this section.

Chapter 3

First Order Logic

It is logical to conclude that "Julie is a human" and "Laura is a human" from the general statement that "All women are human"? In Propositional logic, we are unable to model this deduction. Predicate logic is a formal system modelling these derivations.

3.1 Language

The syntax of predicate logic is less homogenous, for our language must contain nouns, like "Julie" and "Laura", which are the things we talk about, and separate words we apply to nouns, obtaining truth values. These are known as **terms** and **quantifiers** respectively.

Terms should model both definite nouns, such as "Julie" and "Laura", as well as variables, such as X and Y, which can stand for many definite nouns at once, together with relational nouns, such as "The school X went to", a statement describing a noun which varies in interpretation based on the value of X. Definite nouns are known as **constants**, and relational nouns are known as **functions**. Functions will be separated based on their **arity**, the number of arguments they take. "X's favourite Y"is a '2-ary' function, "X's birthday" is a '1-ary' function. We shall now define the terms formally. Let Λ be a set of variables, Δ a set of constants, and for each n, a set Ψ_n of n-ary functions. The set of **terms** is the smallest set $T(\Lambda,\Delta,\{\Psi_n\})$ such that $\Lambda,\Delta \subset T(\Lambda,\Delta,\{\Psi_n\})$, and if $s_1,\dots,s_n \in T(\Lambda,\Delta,\{\Psi_n\})$, and f is a function in Ψ_n, then $f(s_1,\dots,s_n) \in T(\Lambda,\Delta,\{\Psi_n\})$.

It is fairly easy to construct truth functional statements from these term. In addition to the usual connectives of sentential logic, we also require **predicates**, which are functions of nouns representing a statement about those nouns. For instance, "X is a Human" is a predicate. Predicates, like functions, are separated based on arity. For each n, let Π_n be a set of n-ary predicates. Given $\Lambda, \Delta, \{\Psi_n\}$, and $\{\Pi_n\}$, we shall construct a first order language whose terms are as above. Fix a set of variables Λ, a set of constants Δ, a family of functions $\{\Psi_n\}$, and a family of predicates $\{\Pi_n\}$. An **atomic formula** is a string of the form $P(t_1, \dots, t_n)$, where $P \in \Pi_n$, and $t_1, \dots, t_n \in T(\Lambda, \Delta, \{\Psi_n\})$. Then the first-order language $\text{FO}(\Lambda, \Delta, \{\Psi_n\}, \{\Pi_n\})$ is defined to be the smallest set containing all atomic formulae, which is also closed under the logical operations $\wedge, \vee, \Rightarrow, \neg, \Leftrightarrow$, as in sentential logic, and if $x \in \Lambda$ is a variable, and s is a statement, then $(\forall x : s)$ and $(\exists x : s)$ are formulae in the language. We shall abbreviate the string $(\forall x_1 : (\forall x_2 : \dots (\forall x_n : s) \dots))$ as $(\forall x_1, x_2, \dots, x_n : s)$. Its a bit of faff to verify unique interpretation. Given the readers experience, we leave them to fill in the syntactical details when needed. A modification of the brackets lemma used in the previous chapter will aid in this task.

In sentential logic, one may substitute arbitrary formulas into variables, and the meaning of the statement will not change. In first order logic, things are more complicated. Consider the statement

"there exists X, such that if X is a man, then X is mortal"

First off, we cannot replace the initial X, for when we replace it with a definite noun the statement becomes nonsense. We may replace the other X's, but this changes the meaning of the statement

"there exists X, such that if Laura is a man, then Laura is mortal"

The problem results because the instances of X are faulty. Another case results when we substitute X for Y in the formula

"there is Y such that if X is a man, then Y is a dog"

The resulting substitution is

"there is Y such that if Y is a man, then Y is dog"

We wish to perform these types of substitutions, but in a way which avoids changing the meaning of a statement. Let s be an arbitrary string in a first order language. An occurrence of a variable x is **bound** in s if it occurs in a subformula $(\forall x : w)$ or $(\exists x : w)$. An occurrence is **free** if it is not bound, and a variable y is **free for** x **in** s if x does not occur in any subformula of the form $(\forall y : w)$ or $(\exists y : w)$, where y is a free variable in s. A substitution $w[s_1/x_1, \dots, s_n/x_n]$ is only valid when each x_i is free for s_i. This avoids the interpretation problems above. In the first example, X is bound, so cannot be substituted. In the second X is free, but is not free for Y.

3.2 Interpretation

Languages are defined in terms of the subject manner we wish to study, but may be interpreted in many different ways. For instance, the axioms which define the logic of group theory may be interpreted relative to whichever group we interpret the axioms as agreeing with. We would hope that a statement is true if and only if it is true in every interpretation of the axioms. To begin discussing this, we must precisely define what we mean by interpretation, as formulated by Alfred Tarski.

An **interpretation** M of a first order language $\text{FO}(\Lambda, \Delta, \{\Phi_n\}, \{\Pi_n\})$ is a set U_M, known as the **universe of discourse**, and an interpretation of the relations; for each constant $c \in \Delta$, we have an associated element $c_M \in U_M$, for each function $f \in \Phi_n$, we have a function $f_M : U_M^n \to U_M$, and for each proposition $P \in \Pi_n$, we have an n-ary relation P_M on elements of U_M.

In sentential logic, when we assign a truth value to a set of variables, we may extend the definition of truth to all formulas. When we assign a meaning to each variable in a first order language, we may define a meaning on all terms, and from these meanings, assign truth to statements in the corresponding language. Consider a particular interpretation of a first order language with variables Λ, and consider an assignment $f : \Lambda \to U_M$. Define $f_* : T(\Lambda, \Delta, \{\Psi_n\}) \to U_M$, by the recursive formulation. Let x be an arbitrary variable, c an arbitrary constant, and g an arbitrary formula,

$$f_*(x) = f(x) \qquad f_*(c) = c_M \qquad f_*(g(t_1, \dots, t_n)) = g_M(f_*(t_1), \dots, f_*(t_n))$$

Using this definition, we may define whether a formula is satisfied in a model of a first order theory. We shall now define what it means for an assignment to **satisfy** a formula in an interpretation. For simplicity, write

$f[a/x]$ for the assignment

$$f[a/x](y) = \begin{cases} f(y) & y \neq x \\ a & y = x \end{cases}$$

1. f satisfies $P(t_1,\ldots,t_n)$ if $P_M(f_*(t_1),\ldots,f_*(t_n))$ holds.
2. f satisfies $(\forall x : s)$ if s is satisfied by $f[a/x]$, for all $a \in U_M$. f satisfies $(\exists x : s)$ if there is some $a \in U_M$ such that $f[a/x]$ satisfies s.
3. An assignment f satisfies $s \circ u$ or $\neg s$, where $\circ$ and $\neg$ are logical connectives, if the truth evaluation of s and u is consistant with the connectives.

If M is an interpretation, then a formula s is **valid** for an interpretation, denoted $\vDash_M s$, if s is true under every assignment under M. A statement is false if it is true under no interpretation, or alternatively, if the negation of the statement is true under the interpretation. An interpretation is a **model** for a set of formulas Γ if every formula in Γ is true for the interpretation. We shall now explore some useful properties of interpretations.

Lemma 3.1. *If $\vDash_M s$ and $\vDash_M s \Rightarrow w$, then $\vDash_M w$.*

Lemma 3.2. *If a formula s contains free variables $x_1,\ldots,x_n$, and two assignments f and g agree on the free variables, then f satisfies s if and only if g satisfies s.*

Proof. We shall first verify that if t is a term containing variables $x_1,\ldots,x_n$, on which f and g agree, then $f_*(t) = g_*(t)$. If t is a variable, or a constant, the proof is easy. But then by induction, for an n-ary formula u,we have

$$\begin{aligned} f_*(u(t_1,\ldots,t_n)) &= u_M(f_*(t_1),\ldots,f_*(t_n)) \\ &= u_M(g_*(t_1),\ldots,g_*(t_n)) \\ &= g_*(u(t_1,\ldots,t_n)) \end{aligned}$$

Thus the theorem is verified by structural induction.

If s is an atomic formula $P(t_1,\ldots,t_n)$, then f satisfies $P(t_1,\ldots,t_n)$ if and only if g does, because $f_*(t_i) = g_*(t_i)$. If s is $(\forall x : w)$, and f satisfies s, then $f[a/x]$ satisfies w for all $a \in U_M$. But then $g[a/x]$ agrees on all free variables of $f[a/x]$, so $g[a/x]$ satisfies w by induction. It follows that g satisfies s. The remaining cases are easily shown. $\square$

Theorem 3.3. *If s contains no free variables, then either $\models_M s$ or $\models_M \neg s$.*

Proof. This follows from the fact that any two assignments that agree on the free variables of a formula agree on the satisfiability. Thus if there are no free variables, all assignments agree, and in particular all satisfy the formula or all do not satisfy the formula. □

Lemma 3.4. $\models_M (\exists x : s)$ *if and only if* $\models_M \neg(\forall x : \neg s)$.

Proof. If an assignment f satisfies $(\exists x : s)$, then s is satisfied by some $f[a/x]$. But then f does not satisfy $(\forall x : \neg s)$ for $f[a/x]$ does not satisfy $\neg s$. Conversely, if an assignment f does not satisfy $(\exists x : s)$, then every $f[a/x]$ satisfies $\neg s$. □

Lemma 3.5. $\models_M s$ *if and only if* $\models_M (\forall x : s)$.

Proof. If $\models_M s$, then every assignment f satisfies s, so certainly every $f[a/x]$ satisfies s, and thus f satisfies $(\forall x : s)$, hence $\models_M (\forall x : s)$. Conversely, suppose $\models_M (\forall x : s)$. Then, every assignment satisfies $(\forall x : s)$, and thus in particular satisfies s. □

Let s contain free variables $x_1, \dots, x_n$. The **closure** of s is the string $(\forall x_1, x_2, \dots, x_n : s)$. The above proof shows a formula is satisfies if and only if its closure is.

Theorem 3.6. *Consider a form of sentential logic, whose variables are all atomic formulas, and formulas of the form $(\forall x : s)$, and $(\exists x : s)$. Then if a statement is a tautology, then it is satisfied under all interpretations.*

Proof. The connectives of predicate logic are exactly the connectives of sentential logic once we hide away the existential and universal quantifiers. If the statement is a tautology, then regardless of how we interpret the formula, the statement will be satisfied. □

Lemma 3.7. *If t and u are terms, x is a variable, and f is an assignment, then*

$$f[f_*(u)/x]_*(t) = f_*(t[u/x])$$

Proof. If t is a variable unequal to x, or t is a constant, then

$$f[f_*(u)/x]_*(t) = f(t) = f_*(t[u/x])$$

If $t = x$, then

$$f[f_*(u)/x]_*(t) = f_*(u) = f_*(t[u/x])$$

For a structural induction, let $t = g(t_1, \ldots, t_n)$. Then

$$\begin{aligned} f[f_*(u)/x]_*(t) &= g_M(f[u/x]_*(t_1), \ldots, f[u/x]_*(t_n)) \\ &= g_M(f_*(t_1[u/x]), \ldots, f_*(t_n[u/x])) = f_*(t[u/x]) \end{aligned}$$

Thus the theorem holds in general. ☐

Lemma 3.8. *f satisfies $s[u/x]$ if and only if $f[f_*(u)/x]$ satisfies s.*

Proof. If s is $P(t_1, \ldots, t_n)$, then $s[u/x] = P(t_1[u/x], \ldots, t_n[u/x])$, and

$$P_M(f_*(t_1[u/x]), \ldots, f_*(t_n[u/x])) = P_M(f[f_*(u)/x]_*(t_1), \ldots, f[f_*(u)/x]_*(t_n))$$

Which shows that f satisfies $s[u/x]$ if and only if $f[f_*(u)/x]$ satisfies s. If s is formed by standard sentential connectives, the theorem is trivial. If $s = (\forall y : w)$, where $y \neq x$, then $s[u/x] = (\forall y : w[u/x])$, and by definition, f satisfies $s[u/x]$ if and only if $f[a/y]$ satisfies $w[u/x]$ for all a, which by induction implies that $f[f_*(u)/x][a/y]$ satisfies w for all a, so $f[f_*(u)/x]$ satisfies s. Similar results hold if s's primitive connective is the existential quantifier. ☐

Theorem 3.9. *For any formula s and term t free for x, $\vDash_M (\forall x : s) \Rightarrow s[t/x]$*

Proof. Let f be an assignment satisfying $(\forall x : s)$. Then $f[f_*(t)/x]$ satisfies s, so f satisfies $s[t/x]$. ☐

Theorem 3.10. *If s does not contain x as a free variable, then*

$$\vDash_M (\forall x : s \Rightarrow w) \Rightarrow (s \Rightarrow (\forall x : w))$$

Proof. If f satisfies $(\forall x : s \Rightarrow w)$ and s, then $f[a/x]$ satisfies $s \Rightarrow w$, and since s does not contain x as a free variable, $f[a/x]$ also satisfies s, so f satisfies $(\forall x : w)$. ☐

Given a first order language F, together with a specific interpretation with universe of discourse U_M, we enhance the first order language we are discussing to include U as constants. Such an extension is denoted as

$F(U_M)$. U_M can be extended canonically to interpret $F(U_M)$, in the obvious manner. This allows us to discuss formulas of the form

$$\models_M s[u_1/x_1,\ldots,u_n/x_n]$$

Where $u_1,\ldots,u_n \in U$ are formulas containing elements of U. If the free variables of a formula s are $x_1,\ldots,x_n$, then the set of $u_1,\ldots,u_n$ such that M models $s[u_1/x_1,\ldots,u_n/x_n]$ will be called the relation relative to s.

A statement is **logically valid** if it is true under all possible interpretations. A statement is **satisfiable** if it is true under at least one interpretation, and **contradictory** if it is false under every interpretation. A set of statements is satisfiable if they are all true under a single interpretation. A statement s is a **logical consequence** of a set of statements Γ if every interpretation which satisfies every statement of Γ also satisfies s.

What we have argued in this chapter is that certain rules preserve logical validity. Thus they are valid for a formal argument in predicate logic. In the next section, we will formally introduce these rules, and in the system, prove they are all the formulae needed to prove anything valid about predicate logic.

3.3 First Order Formal Systems

We have the syntax and semantics to begin discussing formal systems in the first-order languages. We would like to find axioms which only find theorems satisfied by every model of the system, a **sound** axiom system. Even better, if we can find **complete** axioms, which can proved anything satisfied by every model of the system. For simplicity, we consider only formulae in the connectives $\forall$, $\neg$, and $\Rightarrow$, since all other formulae are equivalent to formulas formed by these connectives. We shall use an axiom consisting of five schemata, the original (A1), (A2), and (A3) found in predicate logic, as well as two new first order schema (A4) and (A5). Let s and w be formula, and t a term substitutable for a variable x. Then (A4) is

$$(\forall x : s) \Rightarrow s[t/x]$$

provided that t is free for substitution for x in s, and (A5) is

$$(\forall x : s \Rightarrow w) \Rightarrow (s \Rightarrow (\forall x : w))$$

where s contains no free occurences of x. Our rules of inference are modus ponens (MP), to infer w from s and $s \Rightarrow w$, as well as universal generalization (UG), inferring $(\forall x : s)$ from s. As in predicate logic, we shall let $\vdash s$ stand for s is a theorem of the system. If Γ is a set of formulae, we shall let $\Gamma \vdash s$ state that s is provable assuming Γ are axioms.

The rule of universal generalization is very subtle. One cannot put it into an axiomatic schema

$$s \Rightarrow (\forall x : s)$$

for this statement is not sound in all models of predicate logic.

Example. *Consider the formula $P(x) \Rightarrow (\forall x : P(x))$. Take a model M whose universe consists of two elements a and b, and let P_M be the relation only satisfied by a. Then, under the assignment $f(x) = a$, $P(x)$ is satisfied, but $(\forall x : P(x))$ is not. Thus $P(x) \Rightarrow (\forall x : P(x))$ is not semantically valid, and hence not provable in first order logic.*

Nonetheless, if we can *prove* s, universal generalization allows us to conclude that $(\forall x : s)$ is also valid; if s contains x as a free variable, it is only an inbetween statement which we use to eventually conclude that $(\forall x : s)$ is true. It is this little annoyance which lead Moses Schönfinkel and Haskell Curry to invent combinatory logic, which is a logical system designed to avoid quantified variables. Nonetheless, common arguments are most easily adapted to our model of first order logic, so for the time being we will stick to this system.

From the theorems we proved about first order semantics, we already know that this axiom system is sound, for each axiom is valid under all interpretations, and our deductions preserve truth. We shall, after some work, conclude this system is complete – one may prove a theorem if and only if it is valid under all interpretations of the theory. To begin with, we notice our system includes (A1), (A2), (A3), and (MP), all the rules of propositional calculus, which justifies a very useful property of our new formal system.

Theorem 3.11. *Let s be a formula in first order logic, and replace it with a corresponding formula in sentential logic by replacing all the main occurences of the quantifiers $(\forall x : s)$ with variables separate from the rest of the variables occuring in s. Then if the formula obtained is a tautology, s is provable in first order logic.*

We shall denote an application of the deduction theorem by (TAUT). There are many useful schemas obtained from this, that are trivial to prove.

- Negation Elimination (NE): $\vdash \neg\neg s \Rightarrow s$
- Negation Introduction (NI): $\vdash s \Rightarrow \neg\neg s$
- Conjunction Elimination (CE): $\vdash s \wedge w \Rightarrow s$, $\vdash s \wedge w \Rightarrow w$.
- Conjunction Introduction (CI): $\vdash s \Rightarrow (w \Rightarrow (s \wedge w))$.

one just makes an application of the tautology theorem.

Example. *If t is free for x in s. Then $s[t/x] \Rightarrow (\exists x : s)$ is a theorem of first order logic, a formalization of existential introduction.*

$\vdash \mathsf{s}[\mathsf{t}/\mathsf{x}] \Rightarrow \neg(\forall \mathsf{x} : \neg \mathsf{s})$

1.	$(\forall \mathsf{x} : \neg \mathsf{s}) \Rightarrow \neg \mathsf{s}[\mathsf{t}/\mathsf{x}]$	*(A4)*
2.	$((\forall \mathsf{x} : \neg \mathsf{s}) \Rightarrow \neg \mathsf{s}[\mathsf{t}/\mathsf{x}]) \Rightarrow (\mathsf{s}[\mathsf{t}/\mathsf{x}] \Rightarrow \neg(\forall \mathsf{x} : \neg \mathsf{s}))$	*(TAUT)*
3.	$\mathsf{s}[\mathsf{t}/\mathsf{x}] \Rightarrow \neg(\forall \mathsf{x} : \neg \mathsf{s})$	*(1),(2),(MP)*

Without the tautology theorem, this deduction would be much longer. It will be useful in further proofs to note that we never applied (UG) to any formulae. We denote an application of this proof by (EI).

The deduction theorem cannot be carried directly over to first order logic, for assumptions in proofs and premises in logical statements are interpreted differently in certain circumstances. $s \vdash (\forall x : s)$ is always true for any statement s, yet $\vdash s \Rightarrow (\forall x : s)$ is not always a theorem.

The problem is, of course, the free variables again causing trouble. In a proof $(s_1, \dots, s_n)$ of $\Gamma \vdash s_n$, we say s_i **depends on** $w \in \Gamma$ if s_i is w, and is obtained as an axiom, or s_i is inferred from s_j and s_j depends on w.

Theorem 3.12. *Suppose that $\Gamma, s \vdash w$, and there is a proof of w from s which involves no application of universal generalization on a formula which depends on w, by a variable x which is free in w. Then $\Gamma \vdash s \Rightarrow w$.*

Proof. We perform induction on the length of proofs. If s can be proved in one statement, then s is either an instance of an axiom, or an element of

$\Gamma \cup \{w\}$. If $s \neq w$, then $\Gamma \vdash s$ since the proof is equally valid here. If $s = w$, then $w \Rightarrow w$ is a tautology, so $\Gamma \vdash w \Rightarrow s$.

Suppose we have a proof $(s_1, \ldots, s_{n+1})$. By induction, for each s_i, we have $\Gamma \vdash (w \Rightarrow s_i)$. If s_{n+1} is an axiom, an element of Γ, or equal to w, we may apply the last paragraph. Suppose s_{n+1} was inferred by modus ponens from s_i and s_j, where s_j has the form $s_i \Rightarrow s_{n+1}$. Then $\Gamma \vdash (w \Rightarrow s_i)$, and $\Gamma \vdash (w \Rightarrow (s_i \Rightarrow s_{n+1}))$ by induction. The statement

$$u = ((w \Rightarrow (s_i \Rightarrow s_{n+1})) \Rightarrow (w \Rightarrow s_i)) \Rightarrow (w \Rightarrow s_{n+1})$$

is a tautology, so $\Gamma \vdash u$, and by modus ponens, we find $\Gamma \vdash (w \Rightarrow s_{n+1})$. Otherwise s_{n+1} is of the form $(\forall x : s_i)$, obtained from some s_i by universal generalization. By induction, $\Gamma \vdash w \Rightarrow s_i$, so $\Gamma \vdash (\forall x : w \Rightarrow s_i)$. By assumption, x does not occur as a free variable of w, so we have the axiom $(\forall x : w \Rightarrow s_i) \Rightarrow (w \Rightarrow (\forall x : s_i))$, and we conclude $\Gamma \vdash w \Rightarrow (\forall x : s_i)$. $\square$

Example. *We have the theorem* $\vdash (\forall x, \forall y : s)) \Rightarrow (\forall y, \forall x : s)$, *for any statement* s. *To see this, we apply our newly established deduction theorem.*

$\vdash (\forall \mathsf{x}, \forall \mathsf{y} : \mathsf{s}) \Rightarrow (\forall \mathsf{y}, \forall \mathsf{x} : \mathsf{s})$

1. $(\forall \mathsf{x}, \forall \mathsf{y} : \mathsf{s})$
2. $(\forall \mathsf{x}, \forall \mathsf{y} : \mathsf{s}) \Rightarrow (\forall \mathsf{y} : \mathsf{s})$ — *(A4)*
3. $(\forall \mathsf{y} : \mathsf{s})$ — *(1),(2),(MP)*
4. $(\forall \mathsf{y} : \mathsf{s}) \Rightarrow \mathsf{s}$ — *(A4)*
5. s — *(3),(4),(MP)*
6. $(\forall \mathsf{x} : \mathsf{s})$ — *(5),(UG)*
7. $(\forall \mathsf{y}, \forall \mathsf{x} : \mathsf{s})$ — *(6),(UG)*
8. $((\forall \mathsf{x}, \forall \mathsf{y} : \mathsf{s}) \Rightarrow (\forall \mathsf{y}, \forall \mathsf{x} : \mathsf{s})$ — $(1-7), (DT)$

Care needs to be taken in order to ensure these steps are accurate, and do not apply universal generalization on a free variable.

Example. *For any s and w, $\vdash (\forall x : s \Leftrightarrow w) \Rightarrow ((\forall x : s) \Leftrightarrow (\forall x : w))$.*

$(\forall \mathsf{x} : \mathsf{s} \Leftrightarrow \mathsf{w}) \Rightarrow ((\forall \mathsf{x} : \mathsf{s}) \Leftrightarrow (\forall \mathsf{x} : \mathsf{w}))$	
1. $(\forall \mathsf{x} : \mathsf{s} \Leftrightarrow \mathsf{w})$	
2. $(\mathsf{s} \Leftrightarrow \mathsf{w})$	*(A4)*
3. $(\forall \mathsf{x} : \mathsf{s})$	
4. s	*(A4), (3), (MP)*
5. w	*(2),(4),(MP)*
6. $(\forall \mathsf{x} : \mathsf{w})$	*(UG)*
7. $(\forall \mathsf{x} : \mathsf{s}) \Rightarrow (\forall \mathsf{x} : \mathsf{w})$	*(3-6),(DT)*
...	
8. $(\forall \mathsf{x} : \mathsf{s}) \Rightarrow (\forall \mathsf{x} : \mathsf{w})$	
9. $(\forall \mathsf{x} : \mathsf{s}) \Leftrightarrow (\forall \mathsf{x} : \mathsf{w})$	
10. $(\forall \mathsf{x} : \mathsf{s} \Leftrightarrow \mathsf{w}) \Rightarrow ((\forall \mathsf{x} : \mathsf{s}) \Leftrightarrow (\forall \mathsf{x} : \mathsf{w}))$	

The proof involved in the elipses is exactly the same as (3) through (6).

A useful theorem is an immediate consequence of the deduction theorem, left to the reader.

Theorem 3.13. *If s has no free occurrences of y, then*

$$\vdash ((\forall x : s) \Leftrightarrow (\forall y : s[y/x]))$$

The next theorem is more involved, but very useful. We define substitution on formulas in the following way. Given formulae s, w, u, we define the formula $s[w/u]$, obtained from swapping u with w, by the base case $u[w/u] = w$, and the recursive case by delving into subformulas, *provided the formula isn't just w.*

Theorem 3.14. *Let $x_1, \ldots, x_n$ be all free variables of s that occurs as a bound variable in u or w. Then*

$$(\forall x_1, \ldots, x_n : w \Leftrightarrow u) \Rightarrow (s \Leftrightarrow s[w/u])$$

The theorem also holds if $s[w/u]$ is replaced with a formula obtained only be swapping some (but perhaps not all) occurences of u.

Proof. We prove, like always, by structural induction. If no occurrences are swapped, we are left with the formula

$$(\forall x_1,\dots,x_n : w \Leftrightarrow u) \Rightarrow (s \Leftrightarrow s)$$

which is a tautology, hence trivial. Thus we may assume that u does occur in s. If s is an atomic formula, then we are left with the case that $s = u$, and then

$$(\forall x_1,\dots,x_n : w \Leftrightarrow u) \Rightarrow (w \Leftrightarrow u)$$

is an instance of (A4). $\square$

It is also useful in mathematics to make arguments of the following form. Suppose we have a theorem of the form $(\exists x : s)$. We introduce a new constant c, which is not used in any axiom of the formal system, and consider the formula $s[c/x]$. If we end up with a formula $w[c/x]$, we conclude that $(\exists x : s)$ implies $(\exists x : w)$. Though our system is not capable of expressing these arguments, it is satisfying to know that such arguments do not increase the amount of theorems one may prove in first order logic. Temporarily, we shall write $\vdash_C s$ for a proof of this form. Formally, we write $\vdash_C s$ if there is a sequence of formulas $(s_1,\dots,s_n)$ such that each s_i is an axiom, is inferred by (MP) or (UG) from a previous formula, or there is a preceding formula $s_j = (\exists x : w)$, where $s_i = w[c/x]$, and c is a new constant which does not occur in any prior formulae or explicitly in axioms of the formal system. We require that no application of (UG) is made by a variable free in some $(\exists x : s)$ by which the new rule is applied. Finally, we require that s_n does not contain any of the constants introduced by the final rule. To prove that this method can be applied to our formal system, we require a certain formula, proved now, and integral to the proof that $\vdash_C$ is redundant.

Example. *If x is not free in w, $((\exists x : s) \Rightarrow w) \Leftrightarrow (\forall x : s \Rightarrow w)$.*

$((\exists \mathsf{x} : \mathsf{s}) \Rightarrow \mathsf{w}) \Leftrightarrow (\forall \mathsf{x} : \mathsf{s} \Rightarrow \mathsf{w})$	
1. $(\exists \mathsf{x} : \mathsf{s}) \Rightarrow \mathsf{w}$	
2. s	
3. $(\exists \mathsf{x} : \mathsf{s})$	*(EI)*
4. w	*(1),(3),(MP)*
5. $\mathsf{s} \Rightarrow \mathsf{w}$	*(2-4),(DT)*
6. $(\forall \mathsf{x} : \mathsf{s} \Rightarrow \mathsf{w})$	*(UG)*
6. $((\exists \mathsf{x} : \mathsf{s}) \Rightarrow \mathsf{w}) \Rightarrow (\forall \mathsf{x} : \mathsf{s} \Rightarrow \mathsf{w})$	*(1-6),(DT)*
7. $(\forall \mathsf{x} : \mathsf{s} \Rightarrow \mathsf{w})$	
8. $\mathsf{s} \Rightarrow \mathsf{w}$	*(A4)*
9. $\neg \mathsf{w} \Rightarrow \neg \mathsf{s}$	*(8),(TAUT),(MP)*
10. $(\forall \mathsf{x} : \neg \mathsf{w} \Rightarrow \neg \mathsf{s})$	*(UG)*
11. $\neg \mathsf{w} \Rightarrow (\forall \mathsf{x} : \neg \mathsf{s})$	*(10),(A5),(MP)*
12. $\neg(\forall \mathsf{x} : \neg \mathsf{s}) \Rightarrow \mathsf{w}$	*(11),(TAUT),(MP)*
13. $(\forall \mathsf{x} : \mathsf{s} \Rightarrow \mathsf{w}) \Rightarrow ((\exists \mathsf{x} : \mathsf{s}) \Rightarrow \mathsf{w})$	*(7-12),(DT)*

Theorem 3.15. *If $\vdash_C s$, then $\vdash s$.*

Proof. Let $(s_1, \dots, s_n)$ be a proof of s, and suppose that

$$s_{i_1} = (\exists x_1 : w_1) \quad s_{i_2} = (\exists x_2 : w_2) \quad \dots \quad s_{i_m} = (\exists x_m : w_m)$$

are all existence formulae used in the proof, from which new constants $c_1, \dots, c_m$ are introduced. Certainly

$$w_1[c_1/x_1], \dots, w_m[c_m/x_m] \vdash s$$

By the universal generalization condition on $\vdash_C$, we can apply the deduction theorem to conclude that

$$w_1[c_1/x_1], \dots, w_{m-1}[c_{m-1}/x_{m-1}] \vdash w_m[c_m/x_m] \Rightarrow s$$

In the proof of this statement, replace all instances of c_m with a new variable y_m. This is then still a valid proof (for variables can be operated on in at least the same capacity as constants), hence we obtain

$$w_1[c_1/x_1], \dots, w_{m-1}[c_{m-1}/x_{m-1}] \vdash w_m[y_m/x_m] \Rightarrow s$$

Hence

$$w_1[c_1/x_1],\dots,w_{m-1}[c_{m-1}/x_{m-1}] \vdash (\forall y_m : w_m[y_m/x_m] \Rightarrow s)$$

applying the recently proved example, since we know y_m does not occur in s at all, we conclude

$$w_1[c_1/x_1],\dots,w_{m-1}[c_{m-1}/x_{m-1}] \vdash (\exists y_m : w_m[y_m/x_m]) \Rightarrow s$$

And by induction, we may assume that

$$w_1[c_1/x_1],\dots,w_{m-1}[c_{m-1}/x_{m-1}] \vdash (\exists y_m : w_m[y_m/x_m])$$

which implies, by a particular use of (MP), that

$$w_1[c_1/x_1],\dots,w_{m-1}[c_{m-1}/x_{m-1}] \vdash s$$

And we may recursively prove that $\vdash s$ applies. $\square$

3.4 Completeness Theorem

The completeness theorem is best understood in the context of **First Order Theories**, which are subsets of formulas in a first order system, which we think of as axioms. An **extension** of a theory is just a theory which is a superset of the original theory. If Γ is a theory, we let $\Gamma^{\mathfrak{d}}$ denote the **deductive closure** of Γ, which is the extension consisting of all formulas s such that $\Gamma \vdash s$. Recall that a theory Γ is **consistant** if for any s, we do not have both $\Gamma \vdash s$ and $\neg s \in \Gamma$. A **complete** theory is a theory in which $s \in \Gamma$ or $\neg s \in \Gamma$ always holds. If some theory is consistance or complete, so is its logical closure.

At times, we shall consider extensions which lie in extensions of first order languages, in the sense of adding additional prepositions, constants, and variable symbols to the language. It is simple to see that if $\mathcal{B}$ is a first order language extending $\mathcal{A}$, if Γ is a theory in $\mathcal{B}$, and if $\Gamma_{\mathcal{B}}$ is the deductive closure of Γ in $\mathcal{B}$, $\Gamma_{\mathcal{A}}$ the deductive closure of Γ in $\mathcal{A}$, then $\Gamma_{\mathcal{A}} = \Gamma_{\mathcal{B}} \cap \mathcal{A}$. Surely the left is included in the second. Alternatively, suppose there is a proof $(s_1,\dots,s_n)$ in $\mathcal{B}$ of $\Gamma \vdash s_n$, with $s_n \in \mathcal{A}$. For each variable and constant in the proof which is not in $\mathcal{A}$, swap it out with a unique unused variable in $\mathcal{A}$. Since variables can do everything a constant can do, the proof is still valid. Similarily, swap each proposition in the proof which is not in $\mathcal{A}$ with some proposition in $\mathcal{A}$. This might be a little harder to see, but one verifies by induction on proof length that this works out, and shows that we may prove s_n solely in $\mathcal{A}$. Thus it is consistant to talk of extensions of theories that add new variables without introducing additional information.

Lemma 3.16. *If a theory has a model, then it is consistant.*

Proof. If Γ has a model, and $\Gamma \vdash s$ and $\Gamma \vdash \neg s$, then s and $\neg s$ must both be valid in the model, which is clearly impossible. $\square$

Example. *The theory of groups is based on the language consisting of some variables, the constant e, a $2-ary$ function $*$, and the $2-ary$ preposition $=$ (applied in infix notation), with the axiom system*

$$x * (y * z) = (x * y) * z$$

$$e * x = x \quad x * e = x$$

$$(\forall x, \exists y : x * y = e \wedge y * x = e)$$

together with the equality axioms

$$(x = y) \Rightarrow (y = x) \quad ((x = y) \wedge (y = z)) \Rightarrow (x = z) \quad (x = y) \Rightarrow (x * z = x * y)$$

This set of statements is a first order theory. It is also interesting to note that the theory is **finitely axiomatizable**, *in the sense that the theory Γ is a finite set. Any model of this theory is just a plain old group. The completeness theorem will show that every theorem proved in this formal system is valid in any group. This theory is consistant, but certainly not complete, for we may add the abelian axiom*

$$x * y = y * x$$

and we certainly have groups which do not satisfy this axiom.

Models of algebraic systems are interesting as an example, but it also is used for interesting applications of logic in algebra. For instance, a simple classification of the algebraically closed fields is obtained through model theoretic methods.

Theorem 3.17. *If we cannot prove s or $\neg s$ in a consistant theory Γ, then $\Gamma \cup \{s\}$ is a consistant theory.*

Proof. Suppose that $\Gamma \cup \{s\} \vdash w$ and $\Gamma \cup \{s\} \vdash \neg w$. Without loss of generality, we may assume that s contains no free variables, for the logical closure of $\Gamma \cup \{s\}$ is the same as the logical closure of $\Gamma \cup \{(\forall x_1, \dots, x_n : s)\}$, because of axiom (A4) and (UG). But then, by the deduction theorem,

$$\Gamma \vdash s \Rightarrow w \qquad \Gamma \vdash s \Rightarrow \neg w$$

But then $\Gamma \vdash \neg s$, because

$$((s \Rightarrow w) \wedge (s \Rightarrow \neg w)) \Rightarrow \neg s$$

is a tautology, a contradiction. □

Lemma 3.18 (Lindenbaum)**.** *If* Γ *is a consistant theory, then there is a complete consistant extension of* Γ*.*

Proof. Let $\mathcal{K}$ be the set of all consistant extensions of Γ, ordered by inclusion. If $\mathcal{A}$ is a chain of consistant extensions of Γ, then we claim $\bigcup \mathcal{A}$ is consistant. Suppose that

$$\bigcup \mathcal{A} \vdash s \qquad \bigcup \mathcal{A} \vdash \neg s$$

Then there is a proof $(s_1, s_2, \dots, s_n)$ of s from $\bigcup \mathcal{A}$, and a proof $(w_1, w_2, \dots, w_n)$ of $\neg s$. Since each side is finite, each uses only finitely many axioms, which implies that there is $\Gamma \in \mathcal{A}$ containing all axioms. But then

$$\Gamma \vdash s \qquad \Gamma \vdash \neg s$$

Which implies A is not consistant, a contradiction. By Zorn's lemma, there is a maximal consistant extension Γ. Γ is complete, by the last lemma. If $\Gamma \nvdash \neg s$, then $\Gamma \cup \{\neg s\}$ is consistant, implying $\Gamma \cup \{\neg s\} = \Gamma$, so $s \in \Gamma$, implying $\Gamma \vdash s$. □

A term is **closed** if it has no free variables. A theory Γ is **scapegoat** if for any formula s which only has one free variable x, there is a closed term t for which

$$\Gamma \vdash (\exists x : \neg s) \Rightarrow \neg s[t/x]$$

Scapegoat theorys are useful for proving the completeness theorem, for it is easier to move constants into interpretations than with plain formulas.

Lemma 3.19. *Every consistent theory* Γ *has a consistant scapegoat extension* Γ *which has the same cardinality as* Γ *if* Γ *is infinite, or is denumerable if* Γ *is finite.*

Proof. The deductive closure of Γ has the same cardinality as Γ in the infinite case, or is denumerable in the finite case. Since the deductive closure extends Γ, we may, without loss of generality, assume Γ is deductively closed. Let $\mathcal{X}$ be the set of all variables in all formulas of Γ which have exactly one free variable, and biject them with a set $\mathcal{C}$ disjoint from all preexisting characters in the first order language of Γ. Let c_x be the element of $\mathcal{C}$ in bijection with $\mathcal{X}$. Let $\tilde{\Gamma}$ be the theory obtained from Γ by adding $\mathcal{C}$ as constants to the theory, together with the formulas

$$(\exists x : \neg s) \Rightarrow \neg s[c_x/x]$$

$\tilde{\Gamma}$ is surely a scapegoat by construction. We claim that $\tilde{\Gamma}$ is consistant and scapegoat. It suffices, since every proof is finite, to show that every finite subextension is consistant. Consider any particular variables $x_1, x_2, \ldots, x_n \in \mathcal{X}$, and let $\Gamma_0 = \Gamma$, and Γ_k be the theory obtained from Γ_{k-1} be adding the contant c_{x_k} and the axiom $(\exists x_k : \neg s_k) \Rightarrow \neg s_k[c_x/x]$. We prove this by induction. Suppose Γ_{k-1} is consistant, and Γ_k is inconsistant. Then we may prove all statements in Γ_{k-1} in Γ_k. In particular,

$$\Gamma_k \vdash \neg((\exists x_k : \neg s_k) \Rightarrow \neg s_k[c_{x_k}/x_k])$$

Since this formula is closed (because x is the only free variable in s_k), we may apply the deduction theorem to conclude

$$\Gamma_{k-1} \vdash ((\exists x_k : \neg s_k) \Rightarrow \neg s_k[c_{x_k}/x_k]) \Rightarrow \neg((\exists x_k : \neg s_k) \Rightarrow \neg s_k[c_{x_k}/x_k])$$

Which allows us to conclude (by the tautology $(A \Rightarrow \neg A) \Rightarrow \neg A$), that

$$\Gamma_{k-1} \vdash \neg((\exists x_k : \neg s_k) \Rightarrow \neg s_k[c_{x_k}/x])$$

Hence

$$\Gamma_{k-1} \vdash (\exists x_k : \neg s_k) \qquad \Gamma_{k-1} \vdash s_k[c_{x_k}/x_k]$$

Since c_{x_k} does not occur in $\Gamma_0, \Gamma_1, \ldots, \Gamma_{k-2}$, we may conclude that $\Gamma_{k-1} \vdash s_k[y_k/x_k]$, by replacing all occurences of c_{x_k} in the proof of $s_k[c_{x_k}/x_k]$ by a new variable y_k which doesnot occur in the proof. But then

$$\Gamma_{k-1} \vdash (\forall y_k : s_k[y_k/x_k])$$

so $\Gamma_{k-1} \vdash s_k$, hence $\Gamma_{k-1} \vdash (\forall x_k : s_k)$, contradicting preestablished theorems. Thus we have shown Γ is consistant. □

Lemma 3.20. *Let Γ be a consistant, complete, scapegoat theory. Then Γ has a model M whose universe of discourse is the set of closed terms in Γ.*

Proof. For each constant c in the language, let $c^M = c$. For each function f, let $f^M(t_1,\dots,t_n) = f(t_1,\dots,t_n)$ (by assumption, each t_i is closed). For each predicate P, let $(t_1,\dots,t_n) \in P^M$ if and only if $\vdash P(t_1,\dots,t_n)$. We shall show that $M \vDash s$ if and only if $\Gamma \vdash s$, for any closed term s. This implies that M is a model of Γ, for then, if w is any axiom of Γ, then $\Gamma \vdash (\forall x_1,\dots,x_n : w)$, hence $M \vDash (\forall x_1,\dots,x_n : w)$, which occurs if and only if $M \vDash w$. Thus M models all axioms. We will prove our statement by structural induction.

1. If s is $P(t_1,\dots,t_n)$, the statement is trivial by construction.

2. Suppose s is $\neg w$. If $M \vDash w$, then by induction $\Gamma \vdash w$, which implies by consistancy that Γ cannot prove $\neg w$, and M cannot satisfy $\neg w$. Conversely, if $M \vDash \neg w$, then $M \nvDash w$, so the theorem follows again by consistancy.

3. If s is $w \Rightarrow u$, since s is closed, w and u are closed. If $M \nvDash w$, then $M \vDash w \Rightarrow u$, and by induction $\Gamma \nvdash w$, so $\Gamma \vdash \neg w$, and then $\Gamma \vdash w \Rightarrow u$ by tautology. The remaining cases are again treated by tautology and completeness.

4. If s is $(\forall x : w)$, then either w is closed, or w has a single free variable. If w is closed, the statement follows from tautologies and completeness fairly easily. We shall treat the other case in detail. Suppose that $M \vDash (\forall x : w)$, yet $\Gamma \nvdash (\forall x : w)$. Thus $\Gamma \vdash \neg(\forall x : w)$. Since Γ is scapegoat, there is a constant c such that $\Gamma \vdash \neg w[c/x]$, from which we conclude $M \vDash \neg w[c/x]$ by induction, contradicting that $M \vdash (\forall x : w)$. Conversely, suppose $M \nvDash (\forall x : w)$, yet $\Gamma \vdash (\forall x : w)$. Then $\Gamma \vdash w[t/x]$ for any term t, so by induction, $M \vDash w[t/x]$ for all closed terms t. Let f be an assignment on M such that which does not satisfy w. Let $f(x) = t$. Since t is a closed term in the interpretation, $f_*(t) = t$, and $f = f[f_*(t)/x]$, so a previous lemma implies that f cannot satisfy $w[t/x]$. Thus $M \nvDash w[t/x]$, a contradiction.

We have addressed all cases, so our proof is complete. □

Theorem 3.21. *Every consistant theory has a model whose cardinality is the same as the theory itself, unless the theory is finite, in which the model is denumerable.*

Proof. Let Γ be a consistant theory. Extend Γ to a consistant, complete, scapegoat theory Γ', which is denumerable if Γ is finite or denumerable, or else Γ has the same cardinality as Γ'. But then Γ' has a model consisting of closed terms in Γ', whose cardinality is the same as the Γ. $\square$

We are now ready for the utmost regarded "Gödel's completeness theorem". Once you know it, you can brag to all your friends that you understand it...

Theorem 3.22 (Gödel (Our proof is Henkin's))**.** *A formula is provable in a theory if and only if it is true under all interpretations.*

Proof. Let Γ be a theory. Without loss of generality, assume Γ is consistant. If $\Gamma \vdash s$, then we have already shown s is true in all models. If $\Gamma \nvdash s$, then $\Gamma \cup \{\neg s\}$ is consistant, so $\Gamma \cup \{\neg s\}$ has a model M, but then $M \models \neg s$, so $M \not\models s$, and M is a model for Γ. $\square$

Thus syntax and semantics coincide. We actually proved something much stronger, that a formula is provable in a theory if and only if it is true under all interpretations whose cardinality is that theories cardinality, or denumarable if the theory is finite. Nonetheless, we find our proof is non-constructive. We did not construct a formal proof given that the formula was true under all interpretations. This foreshadows the general result that there is no constructive procedure for generating a proof of a the formula, proved by Gödel a decade later.

3.5 The Compactness Theorem

Theorem 3.23 (The Compactness Theorem)**.** Γ *is a consistant theory if and only if every finite subset of* Γ *is consistant. Conversely,* Γ *has a model if and only if every finite subset of* Γ *has a model.*

Proof. If Γ is consistant, than a finite subset is consistant. Conversely, suppose Γ is inconsistant, and consider proofs of two statements $\Gamma \vdash s$ and $\Gamma \vdash \neg s$. These proofs only use finitely many axioms in Γ, so there is some finite subset Δ of Γ such that $\Delta \vdash s$ and $\Delta \vdash \neg s$, so some finite subset is inconsistant. $\square$

Consider the space of all consistant theories in a first order logic. Consider the family

$$U_s = \{\Gamma : \Gamma \vdash s\}$$

Then U_s defines a basis for a topology, for $U_s \cap U_w = U_{s \wedge w}$. A net $\{\Gamma_i\}$ converges to Γ in this topology if and only if every statement provable in Γ is eventually provable in the net. Every element of the basis is clopen, for $U_s^c = U_{\neg s}$. The compactness theorem is equivalent to the set $\mathcal{C}$ of complete, consistant theories being topologically compact, for inconsistant theories correspond to open covers of $\mathcal{C}$. If Γ is an inconsistant theory, then

$$\{U_{\neg s} : s \in \Gamma\}$$

is a cover of $\mathcal{C}$. This follows for any complete theory not in any $U_{\neg s}$ must be able to prove every element of Γ, and therefore be inconsistant. Conversely, suppose $\mathcal{U}$ is an open cover of Γ by sets of the form U_s. Then $\{\neg s : U_s \in \mathcal{U}\}$ is inconsistant, for otherwise it is contained in a complete, consistant, deductively closed theory Γ, and $\Gamma \notin U_{\neg s}$ for any $U_{\neg s} \in \mathcal{C}$.

Now suppose $\mathcal{C}$ was compact, and let Γ be an inconsistant theory. Then

$$\{U_{\neg s} : s \in \Gamma\}$$

forms a cover of $\mathcal{C}$. Thus $U_{\neg s}$ has a finite subcover. This corresponds to a finite inconsistant subtheory of Γ. We shall prove $\mathcal{C}$ is compact using the compactness theorem. Let $\mathcal{U}$ be an open cover of $\mathcal{C}$. Without loss of generality, assume each element of $\mathcal{U}$ is of the form U_s. Then $\Gamma = \{\neg s : U_s \in \mathcal{U}\}$ is an inconsistant theory, and therefore has a finite inconsistant subtheory, which corresponds to a finite subcover of $\mathcal{C}$.

Similarily, consider the class $\mathcal{M}$ of all models of a first order system, and take a topology on it by considering the family

$$V_s = \{M \in \mathcal{M} : M \models s\}$$

of open neighbourhoods. Then $M_i \to M$ only when any formula s satisfied by M is eventually satisfied in M_i. For any model M, define $\text{Th}(M)$ to be the set of all *closed* formulas s for which $M \models s$. Then $\text{Th}(M)$ is a consistant theory, which we claim is also complete. Let w be any formula, and let w' be its closure. Then either $M \models w'$ or $M \models \neg w'$, which implies either w or w' is in $\text{Th}(M)$, but then either $\text{Th}(M) \vdash w$ or $\text{Th}(M) \vdash \neg w$, by universal elimination. We claim that the map $\text{Th} : M \mapsto \text{Th}(M)$, from $\mathcal{M}$ to $\mathcal{C}$, is continuous. Given a formula s, consider all M such that $\text{Th}(M) \vdash s$. If s' is the closure of s, then $U_s = U_{s'}$, so we may assume s is closed. But then either $M \models s$ or $M \models \neg s$, and since $\text{Th}(M)$ is consistant, we must have $M \models s$. If $M \models s$, then obviously $\text{Th}(M) \vdash s$, so

$$\text{Th}^{-1}(U_s) = V_s$$

We know Th is surjective, but the next section will show that it is impossible for Th to be injective, unless we limit our models to only having a certain cardinality. Since $\text{Th}(U_s) = V_s$ for each closed formula s, the map is also open, implying $\mathcal{M}$ is compact. A useful corollary is that

Theorem 3.24. *Any sequence of models $M_i \in \mathcal{M}$ contains a convergent subsequence.*

This theorem has interesting consequences. First, note that if Γ is a theory, then the set of all models of a theory Γ is closed in $\mathcal{M}$, since the set of all models M which model Γ are also all models such that $\text{Th}(\Gamma)$ proves all of Γ, which is

$$\text{Th}^{-1}\left(\bigcap_{s \in \Gamma} U_s\right)$$

and each U_s is closed, since its complement is open.

Example. *Supplement the standard definition of $\mathbf{N}$ by adding an additional constant c to the theory. Technically, this increases the number of models of the theory, but any model of this theory can be restricted to an interpretation of Peano arithmetic. For each $n \in \mathbf{N}$, let $\mathbf{N}_n$ be the model of extended peano arithmetic consisting of the standard $\mathbf{N}$, but with c interpreted as n. By compactness, there is a subsequence $\mathbf{N}_{n_i}$ converging to some theory $\mathbf{N}_{n_*}$ of extended peano arithmetic, where n_* is the interpretation of c in this model. For each k, since $M_{n_i} \models c > k$ for all i sufficiently large, $M_{n_*} \models c > k$ for all k, so $n_* > k$. Thus M_{n_*} is a peano model which has 'infinitely large' elements.*

Alternatively, we may consider the theory

3.6 Skolem-Löwenheim

Theorem 3.25 (Skolem-Löwenheim)**.** *Any theory with a model has a model whose cardinality is the same as the theory, or is denumerable if the theory is finite.*

Proof. If a theory has a model, then it is consistant, from which the statement follows from Lindenbaum's lemma. □

We are actually able to prove a much strong proposition.

Corollary 3.26. *If ω is an infinite cardinality greater than or equal to the cardinality of a consistant theory Γ, then there is a theory of size ω.*

Proof. Let M be a model of Γ. Fix $x \in U^M$. Extend U^M to a set U^N of cardinality ω. Each new element will behave exactly like x. We define $f^N(t_1, \dots, t_n) = f^M(t'_1, \dots, t'_n)$, where $t'_k = t_k$ if $t_1 \in U^M$, else $t'_k = x$. Similarily, let $(t_1, \dots, t_n) \in P^N$ if and only if $(t'_1, \dots, t'_n) \in P^M$. Define constants the same as constants are defined in U^M. Then N is a model of Γ of cardinality ω. □

This theorem appears to contradict various classical results, such as the fact that any complete, ordered fields are isomorphic (and thus **R** should be 'the only model' of such fields, which contradicts the cardinality argument). Remember that first order theories are only a model of real mathematics, and thus do not sufficiently encapsulate all mathematical proofs. We therefore learn from the Löwenheim theorem that one cannot prove that **R** is the only complete ordered field, using only the principles of first order logic.

The Skolem Löwenheim theorem holds because first order logic is not sufficiently powerful enough to distinguish 'infinities'. The semantics cannot sufficiently describe what it means for two elements to be equal, so we can hide various model-theoretic 'elements' of a theory inside a single semantic element of a theory. Perhaps, if we add additional equality semantics, can we prevent the Lowenheim theorem from occuring.

3.7 Theories with Equality

We now specialize our study, adding additional axioms that occur in almost every useful first order model. We shall say a first order system possesses equality if the theory has a binary predicate $=$, possessing the additional axioms (A6),

$$x = x$$

and (A7),

$$(x = y) \Rightarrow (s \Rightarrow s')$$

for any variables x and y, and formulae s, where s' is obtained from s by swapping some numbers of occurences of x with occurences of y.

Example. *In any theory with equality, $t = t$ for any term t. This follows from $(A6)$ by substitution. Similarily, $t = s \Rightarrow s = t$ holds.*

$$
\begin{array}{lll}
 & (\mathsf{t} = \mathsf{s} \Rightarrow \mathsf{s} = \mathsf{t}) & \\
 & \quad 1.\ (\mathsf{x} = \mathsf{y}) & \\
 & \quad 2.\ (\mathsf{x} = \mathsf{y}) \Rightarrow ((\mathsf{x} = \mathsf{x}) \Rightarrow (\mathsf{y} = \mathsf{x})) & (A7) \\
 & \quad 3.\ (\mathsf{x} = \mathsf{x}) \Rightarrow (\mathsf{y} = \mathsf{x}) & (1),(2),(MP) \\
 & \quad 4.\ (\mathsf{x} = \mathsf{x}) & (A6) \\
 & \quad 5.\ (\mathsf{y} = \mathsf{x}) & (3),(4),(MP) \\
 & 6.\ (\mathsf{x} = \mathsf{y}) \Rightarrow (\mathsf{y} = \mathsf{x}) & (1\text{-}6),(DT) \\
 & 7.\ (\forall \mathsf{x}, \mathsf{y} : (\mathsf{x} = \mathsf{y}) \Rightarrow (\mathsf{y} = \mathsf{x})) & (UG) \\
 & 8.\ (\mathsf{t} = \mathsf{s} \Rightarrow \mathsf{s} = \mathsf{t}) & (7),(A4),(MP)
\end{array}
$$

We shall also need the theorem $t = s \Rightarrow (s = r \Rightarrow t = r)$, which is an instance of (A7) after some universal generalization and specification.

We have already seen the theory of groups as a first order theory with equality. Here is another example.

Example. *The theory of fields is built on a first order theory with constants 0 and 1, and functions $+$ and $\cdot$. The new axioms (in addition to the axioms of equality) are*

$$x + (y + z) = (x + y) + z \qquad x + 0 = x \qquad (\forall x, \exists y : x + y = 0)$$

$$x + y = y + x \qquad x \cdot (y \cdot z) = (x \cdot y) \cdot z \qquad x \cdot (y + z) = (x \cdot y) + (x \cdot z)$$

$$x \cdot y = y \cdot x \qquad x \cdot 1 = x \qquad x \neq 0 \Rightarrow (\exists y : x \cdot y = 1) \qquad 0 \neq 1$$

If we add another binary predicate symbol $<$, and add axioms

$$x < y \Rightarrow x + z < y + z \qquad x < y \wedge 0 < z \Rightarrow x \cdot z < y \cdot z$$

Then we obtain the theory of ordered fields.

Example. *One of the most important historical axiom systems was a system for geometry. The new predicates of the system are I, P, and L. $P(x)$ means x*

is a point, $L(x)$ means x is a line, and $I(x,y)$ means x lies on y (is incident to). The new axioms are

$$P(x) \Rightarrow \neg L(x) \qquad I(x,y) \Rightarrow P(x) \wedge L(y) \qquad L(x) \Rightarrow (\exists y \neq z : I(y,x) \wedge I(z,x))$$

$$P(x) \wedge P(y) \wedge x \neq y \Rightarrow (\exists z : L(z) \wedge I(x,z)) \quad (\exists x,y,z : P(x) \wedge P(y) \wedge P(z) \wedge \neg C(x,y,z))$$

where $C(x,y,z)$ is the collinear relation

$$(\exists u : L(u) \wedge L(x,u) \wedge L(y,u) \wedge L(z,u)$$

This is the theory of geometry, which extends to Euclidean, Projection, and Hyperbolic geometry, so that the theory is not complete.

In theories of equality it is possible to define new symbols which are useful for abreviating formulae. We define $(\exists! x : s)$ to means that there is only one element x satisfying s. That is, the symbol is short for

$$(\exists x : s) \wedge (\forall x,y : s[x/x] \wedge s[y/x] \Rightarrow x = y)$$

Similarily, for each integer n, we may define the symbol $(\exists_n x : s)$. $(\exists_1 x : s)$ is the same as $(\exists! x : s)$.

In any model M of a theory with equality, the relation $=^M$ is an equivalence relation. The model M is **normal** if the equivalence relation is trivial. Any model M can be contracted into a normal model M'. Given a model M, we quotient U^M by $=^M$ to obtain $U^{M'}$. For a function f and predicate P, define

$$f^{M'}([t_1],[t_2],\dots,[t_n]) = [f^M(t_1,\dots,t_n)]$$

$$([t_1],[t_2],\dots,[t_n]) \in P^{M'} \quad \text{iff} \quad (t_1,\dots,t_n) \in P^{M'}$$

These definitions are well defined, since the model interprets equality correctly. All axioms are correctly interpreted by the model as well. We obtain an extension to last chapter.

Proposition 3.27. *Any consistant theory of equality has a normal model whose cardinality is less than or equal to the cardinality of the theory in question.*

Proof. Contract any particular model. □

And now, the true Löwenheim-Skolem theorem, since the other proof relied on a big cheat, which we cannot rely on in normal models.

Corollary 3.28 (Löwenheim-Skolem)**.** *Any theory with equality with an infinite normal model has a model whose size is the same as the theory if the theory is infinite, or is countable otherwise.*

Proof. Let Γ be a consistant theory with an infinite normal model M. First, let us treat the case where the cardinality of Γ is infinite, but less than or equal to the cardinality of the theory. Add to Γ new constants c_α, with axioms $c_\alpha \neq c_\beta$ if $\alpha \neq \beta$, forming a new theory Γ'. We claim Γ' is consistant. Suppose that $\Gamma' \vdash s \wedge \neg s$, which is without loss of generality also a formula in the first order theory related to Γ. The proof of $s \wedge \neg s$ only used a finite number of axioms of the form $c_{\alpha_1} \neq c_{\beta_1}, \ldots, c_{\alpha_m} \neq c_{\beta_m}$. Let M be an infinite model of Γ. Since M is infinite, we may extend the model to a model M', with the same universe of discourse, interpreting c_{α_i} and c_{β_i} in such a way that $c_{\alpha_i} \neq c_{\beta_i}$ is satisfied in M. But then M' is a model for

$$\Gamma \cup \{c_{\alpha_1} \neq c_{\beta_1}, \ldots, c_{\alpha_m} \neq c_{\beta_m}\}$$

which therefore must be consistant, a contradiction. Thus Γ' is consistant, and has an infinite normal model. $\square$

3.8 New Function Letters and Constants

Chapter 4

Combinatory Logic

Standard propositional logic formalizes the mathematical process of proof, pincering mathematical statements in a formal system where they can be examined in detail. Rather than looking at statements which can be verified true or false, Combinatory logic instead analyzes higher order functions, and our limitations in defining them. These functions are universally found in mathematics. The statement $(\forall x : \neg P(x) \vee Q(x))$ contains the functions $\neg$, $\vee$, P, Q, and $\forall$, and it is semantically interesting to interpret the statement not as a schema which can be applied to assert truths about objects of some domain of discourse, but instead as a single truth about the logical function obtained by composing the atomic logical operations. Combinatory logic studies the formal representation of these higher order functions, revealing the limitations in the functions we can define.

The primal nature of the substitution operator was discovered early on in the theory of combinatory logic. It is fundamental tool for constructing functions. For instance, if we represent numbers as tallies, so that 1 is $\cdot$, 2 is $\cdot\cdot$, and 7 is $\cdot\cdot\cdot\cdot\cdot\cdot\cdot$, then we can add 3 to any number by substituting it into the expression $\cdot\cdot\cdot x$ for the variable x. Similarily, we may multiply a number by two by substituting it into the equation xx for x. Addition and multiplication is thus a special case of substitution, and we shall find that a more complicated specification of this process will suffice to represent any function. Substitution is so important to combinatory logic that the field is often seen as the formal analysis of substitution.

Combinatorial logic was initially designed to provide a foundation for mathematics, where we can view inference rules as operators on formulas which can be formally analyzed. The operation of substitution plays a subtle role in much of this process, and this encouraged Combinatorial logic forerunner Moses Schönfinkel to introduce substitution as an explicit connective in a formal system. It turns out that unrestricted substitution is an incredibly volatile operator, especially in the context of first order logic. While it provides expressive power, it also opens the floodgate to paradoxes which can easily lead to an inconsistant system. Even if these careless systems are refined, the systems are likely not expressive enough to handle all mathematical concepts. Nonetheless, the formal systems of combinatory logic are sufficiently expressive to describe all realistic forms of computation, and it is this application which makes the theory useful to modern computing science.

4.1 The λ Calculus

The λ-calculus is the most famous formalization used to study combinatory logic. Every formula in the calculus represents an operator over the other formulas in the calculus, and there are only two fundamental connectives we can use to manipulate these operators. The first is **function application**, which takes a function f and applies it to an argument a, denoted $(f\,a)$. The second is **function abstraction**, which transforms a term t into a function $(\lambda x.t)$, which takes some argument a and substitutes it for x in t. We then use substitution rules on terms to express computation, and this is where the real fun of λ calculus begins.

Terms of the λ calculus can be applied arbitrarily to other terms, include the term itself. This distinguishes the view of functions in set theory, where it is impossible to consider a function application $f(f)$ without contradicting the axiom of regularity. Thus terms of the λ calculus model something very different to standard set theoretic functions. To distinguish this form of the calculus from other formal systems where the operators have particular domains, we call this the **untyped** version of the λ calculus. Nonetheless, unrestricted application is not a deficiency in the calculus – it has a reasonable interpretation in mathematics, which we will discuss later, and is required to construct the class of all computable functions.

So lets define the terms of λ calculus formally. First, we consider a symbol set consisting of variables, constants, the abstractor symbol λ, parenthesis, and a period as a separator. We define terms to be the smallest set of strings over these symbols such that

- All variables and constant are terms.
- If M and N are terms, then (MN) are terms.
- If M is a term, and x is a variable, then $(\lambda x.M)$ is a term.

If there are no constants, we call this a **pure** version of the λ calculus, because there are no constants, so every term of the calculus is a pure function. There is essentially no distinction between constants and variables in the λ calculus, except that we can perform λ abstraction on variables, so we won't consider constants in these notes.

We introduce some shorthand to ensure we don't get overwhelmed by parenthesis. First, we let terms associate to the left, so that $N_1 \dots N_n$ is shorthand for the term $(\dots((N_1 N_2) N_3)\dots)N_n)$, and we write $(\lambda x_1 \dots x_n.M)$ for the term $(\lambda x_1.(\lambda x_2.(\dots(\lambda x_n.M)\dots)))$. This shorthand is meant to let of think of a series of λ abstractions as representing a multidimensional function. This is the technique of Currying: we can think of a function which takes two arguments as a function which takes a single argument, and then returns a function which takes another argument to calculate the overall result!

On the set of terms of the λ-calculus, we introduce **reduction rules**, which not only simplify formulas, but act as the formal model of computation required for the theory.

- α **reduction** is a safe way to rename variables. If M is a term, and y is a variable not occuring anywhere in M, then we have a one-step α reduction $(\lambda x.M) \rhd_{\alpha,1} (\lambda y.M[y/x])$. More generally, if M contains a subterm of the form $(\lambda x.N)$, and if M' is the operation formed by replacing an occurence of $(\lambda x.N)$, then we will let $M \rhd_{\alpha,1} M'$. The transitive closure of the $\rhd_{\alpha,1}$ relation will be denoted $\rhd_\alpha$.
- β **reduction** introduces a semantic computational step into the calculus. Given a β **redex**, a term which can be written $(\lambda x.M)N$, we write $(\lambda x.M)N \rhd_{\beta,1} M[N/x]$, provided that N is 'safe to substitute' for x in M. This means that every free occurrence of x in M occurs in a subterm of the form $(\lambda y.M_0)$, where y is a free variable in N. Similarily, we will $\rhd_{\beta,1}$ be allowed on subterms of a general term, and the transitive closure will be denoted $\rhd_\beta$.

- A relation $\rhd$ satisfies the ξ **rule** if $M \rhd N$ implies $\lambda x.M \rhd \lambda x.N$.
- A relation $\rhd$ is closed under composition if $M_0 \rhd N_0$ and $M_1 \rhd N_1$ implies $(M_0 M_1) \rhd (N_0 N_1)$.

The general reduction relation, which we shall denoted by $\rhd$ or $\rhd_\lambda$, is the smallest transitive relation $\rhd$ containing α and β reduction. It satisfies the ξ rule, and is closed under composition, because α and β reduction are allowed over subterms. Conversely, $\rhd$ is also the smallest transitive relation containing α and β reduction *not* over subterms, closed under composition, and satisfying the ξ rule. As an example, since

$$(\lambda yx.xy)xy \rhd_\alpha (\lambda y_0 x_0.x_0 y_0)xy \rhd_\beta yx$$

we find that $(\lambda yx.xy)xy \rhd yx$. If we enlarge reduction to be the smallest reflexive, symmetric relation, we obtain the notion of equivalence, which we write as $M =_\lambda N$, or if we want to emphasize the proof theoretic aspects, as $\lambda \vdash M = N$.

As $\rhd$ is the smallest transitive relation containing α and β reduction, we obtain an inductive method to prove that a property $R(x, y)$ over the terms of the λ calculus satisfies $R(M, N)$ for any M which reduces to N. It suffices to verify that

- R is transitive.
- $R((\lambda x.M)N, M[N/x])$, where N is safe for substitution for x in M.
- If $R(M_0, M_1)$ and $R(N_0, N_1)$ hold, then $R(M_0 N_0, M_1 N_1)$ also holds.
- If $R(M, N)$ holds, then $R(\lambda x.M, \lambda x.N)$ holds.

because then R is a transitive relation containing α and β reduction, and satisfying the ξ and composition rule, hence R contains the relation of reduction, and so if $M \rhd N$, then $R(M, N)$ necessarily holds. If R is symmetric and reflexive, then we can also conclude that if $M =_\lambda N$, then $R(M, N)$ holds.

It is important to consider how important it is that we only perform β reduction on terms safe for substitution. If not, then we could conclude that

$$(\lambda xy.x)y \rhd_\beta (\lambda y.y)$$

$$(\lambda xy.x)y \rhd_\alpha (\lambda xy_0.x)y \rhd_\beta \lambda y_0.y$$

We would like to interpret reduction as contracting a function definition to a shorter definition which defines an equivalent function. However, the two functions above certainly should not be equivalent. Indeed, the first represents the identity function $f(x) = x$, and the second represents the constant function $g(x) = y$, for some y. Even without a semantic interpretation, we can further use these reductions to conclude that

$$\lambda \vdash M = (\lambda y.y)M = (\lambda y_0.y)M = y$$

so we find that any two terms of the λ calculus are equal. This is clearly not desirable in an actual theory for representing interesting classes of operators.

We say a term M is in β **normal form** if it contains no subterms which form a β redex. This means exactly that the only reductions possible from M are α reductions: If $M \rhd N$, then $M \rhd_\alpha N$. Since α conversion isn't much of an interesting calculation, we view β normal forms as forms which have been completely computed. We can then interpret β normal forms as terms representing algorithms which eventually halt.

Example. *The term $(\lambda x.(\lambda y.yx)z)a$ has normal form za, because*

$$(\lambda x.(\lambda y.yx)z)a \rhd_\beta (\lambda x.zx)a \rhd_\beta za$$

and za does not contain any β redexes.

Example. *The term $\Omega = (\lambda x.xx)(\lambda x.xx)$ has no normal form, because α reduction only changes the term to $(\lambda y.yy)(\lambda z.zz)$, for some variables y and z, and β reduction on any term of this form results in $(\lambda z.zz)(\lambda z.zz)$. Thus the set of all compositions of terms of this form is closed under β and α reduction, and all of them contain a β redex, so we conclude that Ω has no normal form. It is singular because it is the only term up to α conversion which cannot be reduced to some other term.*

There is a more formal way to define the class of all normal forms. We take the following inductive definition:

- All variables are in normal form.

- If M is in normal form, and x is a variable, then xM is in normal form.

- If M is in normal form, then $\lambda x.M$ is in normal form.

Surely any element of this grammar is in normal form. Conversely, if xM is in normal form, then so too is M because the subterms of M cannot be β redexes, and if $(\lambda x.M)$ is in normal form, then so too is M because M is in normal form. Thus the problem of deciding whether a term of the λ calculus is in β normal form can be expressed as a context free grammar, a computable operation. However, the problem of determining whether a term of the λ calculus can be *reduced* to a term in β normal form is undecidable. It is essentially the same problem as proving a Turing machine halts.

4.2 Consistency and Church-Rosser

We can express the λ calculus as an equational theory of logic. However, the theory is not a first order logic, because the terms of the calculus contain variables which may be bound, and this is not possible in vanilla first order logic. In order to get around this, we form a formal theory with only a single predicate, so that the propositions of the system are exactly of the form $M = N$, for some terms M and N, and we consider the reduction rules as inferences from the basic equality axioms. Because the λ calculus is not modelled as a first order theory, the model theory will have to be slightly different. The axioms of the system are

$$\begin{array}{lr} (\alpha) & \lambda x.M = \lambda y.M[y/x] \\ (\beta) & (\lambda x.M)N = M[N/x] \\ (\mathrm{id}) & M = M \end{array}$$

and the inference rules are

$$\frac{M = N}{LM = LN}\ (\mu) \qquad \frac{M = N \quad N = L}{M = L}\ (\tau) \qquad \frac{M = N}{LM = LN}\ (\nu)$$

$$\frac{M = N}{N = M}\ (\sigma) \qquad \frac{M = N}{\lambda x.M = \lambda x.N}\ (\xi)$$

An equation is provable in this axiom system if and only if the equation is true for the λ terms.

Since the λ calculus is equational, there is no way to talk about the standard consistency of the inference rules – there is no such thing as a contradiction, because we don't necessarily have a negation connective. However, the syntax does give rise to some predicates in the metalanguage, those being the equality predicates $\lambda \vdash M = N$. Since formal systems using classical logic are inconsistant precisely when every statement in the language can be proved, we will say that the λ calculus is consistent if *not every* equation can be proved. That is, there is some M and N such that $M = N$ cannot be proved.

The Church-Rosser theorem is the central result to proving the consistency of the λ calculus. It is clear from the result that a β normal form is unique up to α reduction. Thus if two terms M and N are in normal form, and cannot be α converted into one another, then we cannot prove that $M = N$. We surely have two terms which are not α equivalent, so we can conclude that the λ calculus is a consistent theory, as a direct result.

Theorem 4.1 (Church-Rosser for $\triangleright$). *If $M \triangleright M_0$ and $M \triangleright M_1$, then there is a term N such that $M_0 \triangleright N$ and $M_1 \triangleright N$.*

Proof. Annoyingly technical, so I'll prove it some other time. □

The property that the Church-Rosser theorem proves for $\triangleright$ is called **confluence**. There is an analogous result for equality of terms. Note that for any symmetric relation, confluence is trivial, so the theorem has to be strightly stronger than this to be interesting.

Theorem 4.2. *If $\lambda \vdash M = N$, then there is a term L such that $M \triangleright L$ and $N \triangleright L$.*

Proof. First, note that if $\vdash M = N$, then there is $L_1, \dots, L_n$ with $L_0 = M$, $L_n = N$, and either $L_k \triangleright L_{k+1}$ or $L_{k+1} \triangleright L_k$. We shall prove this theorem by induction on n. If $n = 1$, the theorem is trivial, because M is syntactically equal to N. If $n = 2$, then either $M \triangleright N$ or $N \triangleright M$, and then the theorem is just the Church-Rosser theorem for $\triangleright$. Otherwise, by induction, we may assume that there is L with $L_1 \triangleright L$ and $L_{n-1} \triangleright L$. If $L_n \triangleright L_{n-1}$, then $L_n \triangleright L$, hence the theorem is complete. Thus we may assume $L_{n-1} \triangleright L_n$. But then by applying Church-Rosser for $\triangleright$, we conclude that there is K such that $L \triangleright K$ and $L_n \triangleright K$, and this completes the proof, because $L_1 \triangleright L \triangleright K$. □

Essentially, what we have proved is that if R is any transitive confluent relation, and we take the smallest symmetric extension R', then if $R'(x, y)$, then $R(x, z)$ and $R(y, z)$ for some z.

Corollary 4.3. *If $\lambda \vdash M = N$, and N is in β normal form, then $M \triangleright N$.*

Proof. The last theorem shows that $M \rhd L$ and $N \rhd L$ for some term L. But then L is alpha congruent to N, hence $L \rhd N$, and so $M \rhd N$. $\square$

Corollary 4.4. *If $\lambda M = N$, and M and N are both in β normal form, then M and N are α equivalent.*

Terms without a normal form correspond to algorithms which don't terminate. Such terms are essentially meaningly in the λ calculus. But the fact that a term has a normal form does not imply that subterms of the term have a normal form. For instance, $(\lambda x.y)\Omega$ has a normal form y, whereas we know Ω does not have a normal form. This means that a 'meaningful' term may have meaningless subterms. This seems undesirable, so it is of interest to reduce the terms of the λ calculus so that subterms of terms with normal forms have normal forms. Church found the system λI with this property. It is defined in essentially the same way as the standard λ calculus except that when forming the terms of the system, we only let $(\lambda x.M)$ be a term when x is a free variable in M.

4.3 Combinators

An alternative formal system in which to discuss combinatory logic is the theory of combinators. One can see this theory as the subtheory of the λ calculus generated by the closed terms. Suprisingly, this subtheory turns out to be equivalent in expressive power to the entire λ calculus theory. This may seem unreasonable in the context of λ calculus, for we need free variable terms in order to form future functions. However, we can represent λ abstraction via the composition of closed λ terms, so that there is a formal system for combinatory logic with composition as the only connective. We will introduce variables into combinatory logic for convenience, but we note they are not required for most discussions of the calculus, and are much simpler than variables in the λ calculus because there is no way to bind variables.

Combinatory logic was invented a decade before the λ calculus, and can be developed with no mention of Curry's theory at all. One starts with a set of primitive combinators, and a set of variables, and inductively constructs the set of all combinators.

- Every primitive combinator and variable is a combinator.

- If A and B are combinators, then (AB) are combinators.

Equivalence of terms is generated based on substitution rules for the primitive combinators. We associate with each base combinator C a substitution rule of the form $Cx_1 \dots x_n \rhd_A A_C$, where A_C is another term in the λ calculus. We can then define one step reduction as the relation formed by the rules $CB_1 \dots B_n \rhd_{C,1} A_C[B_1/x_1, \dots, B_n/x_n]$, or more generally, if this reduction occurs in a subterm of a combinator. The union of all $\rhd_{C,1}$ is one step reduction $\rhd_1$, and the transitive closure is general reduction $\rhd$. We can then consider equivalences by taking the reflexive symmetric closure, and we denote this by $M =_{\mathrm{CL}} N$, or $\mathrm{CL} \vdash M = N$.

Example. *There are some classical combinators which are universally known.*

$$\begin{array}{ccc} Ix \rhd x & Bxyz \rhd x(yz) & Sxyz \rhd xz(yz) \\ Kxy \rhd x & Cxyz \rhd xzy & Wxy \rhd xyy \\ Mx \rhd xx & B'xyz \rhd y(xz) & Yx \rhd x(Yx) \end{array}$$

Note that all the combinators but the Y combinator have axioms which are formed from the variables in the definition. We call these types of combinators **proper combinators**, *rather than* **improper combinators**.

Combinators can be classified by five features, based on how the substitution relations work. **Identity combinators** are those which operate by reductions of the form $Cx_1 \dots x_n \rhd_C x_1 \dots x_n$. The classical I combinator above is an identity combinator of arity one. The combinator BI satisfies $BIxy \rhd_B I(xy) \rhd_I xy$, hence BI acts as an identity combinator of arity two. An **associator** is a combinator which groups the input terms. The B combinator is an associator, and is the only non-trivial associator with arity three (the other associator would be $Bxyz \rhd (xy)z$, but this is just an identity combinator of arity three). An example of a more complicated combinator is a combinator C with the axiom $Cxyzvw \rhd_C x((yz)vw)$. **Cancellators** remove variables from an input, like $Kxy \rhd_K x$, and **permutators** permutes arguments. Finally, **duplicators**, as expected, duplicate arguments.

Since combinatory logic is an equational theory, we can also talk about consistency. It is often useful to use a sequent calculus for describing the theory. The only new axiom is primitive reduction

$$(\rho) \quad CB_1 \dots B_n = A_C[B_1/x_1, \dots, B_n/x_n]$$

in addition to the axiom $M = M$. We then use the standard inference rules

$$\frac{M = N}{LM = LN}\ (\mu) \qquad \frac{M = N \quad N = L}{M = L}\ (\tau) \qquad \frac{M = N}{LM = LN}\ (\nu)$$

$$\frac{M = N}{N = M}\ (\sigma)$$

One interesting difference between combinatory logic and the lambda calculus is that, because combinatory logic has no variable binding operations, the system can be formalized as a first order equational theory. This will have interesting consequences for the differences between the model theory of combinatory logic and the model theory of the λ calculus.

A **weak redex** is a combinator of the form $AB_1 \ldots B_n$, on which we can perform a reduction. If a term contains no weak redexes, we say it is in **weak normal form**, which is essentially the end result of a calculation. There is a version of the Church Rosser theorem for reduction in combinatory logic, so that $\rhd$ is a confluent operation, and the system of combinatory logic is consistant. That is, weak normal forms are unique if they exist. Furthermore, if $M =_{\mathrm{CL}} N$, there is a combinator L such that $\mathrm{CL} \vdash M \rhd L$ and $\mathrm{CL} \vdash N \rhd L$.

Intuitively, any base term in combinatory logic can be modelled in the λ calculus, because the axioms for primitive combinatory logic are analogous to closed λ abstractions. For instance, the standard combinator axioms

$$\begin{array}{ccc} Ix \rhd x & Bxyz \rhd x(yz) & Sxyz \rhd xz(yz) \\ Kxy \rhd x & Cxyz \rhd xzy & Wxy \rhd xyy \\ Mx \rhd xx & B'xyz \rhd y(xz) & Yx \rhd x(Yx) \end{array}$$

are analogous to

$$\begin{array}{ccc} \lambda x.x & \lambda xyz.x(yz) & \lambda xyz.xz(yz) \\ \lambda xy.x & \lambda xyz.xzy & \lambda xy.xyy \\ \lambda x.xx & \lambda xyz.y(xz) & \lambda xyzv.xy(xvz) \end{array}$$

Conversely, any closed λ term with all the λ terms to one side is analogous to an axiom. The position of the λ terms is important to the theory, which hints at why the syntactic theory of combinatory logic and the λ calculus is different. However, we should expect that sufficiently powerful combinators should lead to a form of logic expressive enough to represent all substitution rules.

We will say a combinatory logic is **combinatorially complete** if, for any axiom rule of the form $Cx_1 \dots x_n = A_C$, there is a combinator C' (not necessarily primitive) such that $C'B_1 \dots B_n \rhd A_C[B_1/x_1, \dots, B_n/x_n]$ for all combinators B_i (we need only show this for a set of variables). The most well known combinatorially complete logic consists of the primitive combinators $\{S, K\}$, but other combinatory basis, like $\{I, B, C, W, K\}$ and $\{I, J, K\}$, exist. We shall actually prove that $\{S, K, I\}$ is combinatorially complete, rather than $\{S, K\}$, but since $SKKx \rhd I$ for all combinators x, the combinatorial completeness of $\{S, K, I\}$ implies the combinatorial completeness of $\{S, K\}$. Our method to do this is to introduce a form of 'lambda abstraction' which forms a part of the metatheoretic language to express all substitution rules.

Given a variable x and a combinator M, we shall define a combinator $[x].M$ such that $[[x].M]y \rhd M[y/x]$ for all combinators y. This will enable us to write arbitrary substitution rules in the calculus.

- If none of the x occur in M, define $[x].M = KM$. Then $KMy \rhd M$, and $M = M[y/x]$, hence $[x].M$ satisfies the required property.

- If none of the x occur in M, defined $[x].Mx = M$. Then $My \rhd My$, and $My = [Mx][y/x]$. This is not necessary, because it is covered by other cases, but leads to a simple formula for λ abstraction.

- Let $[x].x = I$. Then $Iy \rhd y$, and $y = x[y/x]$.

- Otherwise, let $[x].(MN) = S[[x].M][[x].N]$. Then by induction,

  $$S[[x].M][[x].N]y \rhd ([[x].M]y)([[x].N]y) \rhd M[y/x]N[y/x]$$

  and $M[y/x]N[y/x] = (MN)[y/x]$.

Note that $[x].M$ does not contain the variable x. Thus if we extend substitution multidimensionally, so that $[x_1, \dots, x_n].M$ is defined recursively as $[x_1].[[x_2, \dots, x_n].M]$. By induction, we find

$$[[x_1, \dots, x_n].M]x_1 \dots x_n \rhd M$$

And because $[[x_1, \dots, x_n].M$ does not contain the variables x_i, we find by substitution that $[[x_1, \dots, x_n].M]y_1 \dots y_n \rhd M[y/x]$. This uses the general principle that if $M \rhd N$, then $M[y/x] \rhd N[y/x]$. So now, if we have an axiom of the form $Cx_1 \dots x_n \triangleright_C A_C$, we let $B = [x].A_C$. Then we have shown that $By_1 \dots y_n \triangleright A_c[y/x]$, hence $\{K, S\}$ is combinatorially complete. Essentially, this is the main technique to proving a set of combinators is complete, albeit reducing the set to a basis which is already proved complete.

Since we have a form of 'lambda abstraction' for combinators, it is an interesting question to ask whether the ξ rule holds in our calculus. That is, if $\vdash_{\mathrm{CL}} M = N$, then does it necessarily hold that $\vdash_{\mathrm{CL}} [x].M = [x].N$. One difference between combinatory logic and the lambda calculus is that the ξ rule need not hold in combinatory logic.

Example. *If $M = Sxyz$ and $N = xz(yz)$, then $M \triangleright_S N$, yet*

$$[x].M = S(S(S(KS)I)(Ky))(Kz)$$

$$[x].N = S(SI(Kz))(S(Ky)(Kz))$$

and both elements are in normal form, hence they cannot be CL equivalent.

The lack of the ξ rule is normally no problem, but sometimes it does cause issue, which means λ calculus becomes a more applicable theory. Conversely, combinatory logic has a more elegant theory of substitution, so we must make a tradeoff between the two. The ξ rule will return when we analyze the similarities between the formal theory of the λ calculus and the formal theory of combinatory logic.

4.4 Extensionality

An intensional process is one which is defined by its description, whereas an extensional process is one uniquely defined by its inputs and outputs. The functions of set theory are extensional, because two functions are equal if and only if they agree on all inputs. Conversely, the terms of the λ calculus and the combinators of combinatory logic are viewed as intensional descriptions of processes, because they describe the process of substitution, and just because Mx is equivalent to Nx for all inputs x does not imply that M is equivalent to x. Algorithms in computing science are intensional, because two algorithms which solve the same problem are not necessarily viewed as equivalent, especially if the runtime of the algorithms is under scrutiny.

It turns out that the theory of the λ calculus is equivalent to combinatory logic as a formal system, provided we introduce extensionality to the system. Recall that a definition of function is extensional if functions are identified by their inputs and outputs, as in set theory. That is, if $fx = gx$ for all x, then $f = g$. The λ calculus is an intensional theory because this need not hold. For instance, if $f = y$, and $g = (\lambda x.yx)$, then

$$\lambda \vdash gx = (\lambda x.yx)x = yx = fx$$

yet we cannot prove that $g = f$ in the λ calculus, since both terms are in normal form. Similar results hold for combinatory logic – we can define two primitive terms with the same substitution rule, yet the two terms will be unequal in the theory.

We extend the formal theory of the λ calculus to be intensional by adding an additional inference scheme to the proof system. If x is a variable, and M and N are terms not containing x as a free variable, we add the rule

$$\frac{Mx = Nx}{M = N}\ (\zeta)$$

The extended theory is known as the $\lambda\zeta$ calculus, and we write $\lambda\zeta \vdash M = N$ for entailment of the equation theory. We can also consider adding the axiom

$$(\eta) \qquad (\lambda x.Mx) = M$$

where x is not a free variable of M, and we write $\lambda\eta \vdash M = N$ for this formal theory.

Theorem 4.5. *The inference rule ζ is correct in the $\lambda\eta$ theory, in the sense that if $\lambda\eta \vdash Mx = Nx$, and M and N do not contain x as a free variable, then $\lambda\eta \vdash M = N$. Conversely, the axiom η is derivable in the $\lambda\zeta$ theory.*

Proof. The first part of the theorem is proved by the following sequent tree in the $\beta\eta$ calculus.

$$\cfrac{\cfrac{(\lambda x.Mx) = M \quad (\eta)}{M = (\lambda x.Mx)}\ (\sigma) \qquad \cfrac{\cfrac{Mx = Nx}{(\lambda x.Mx) = (\lambda x.Nx)}\ (\xi) \qquad (\lambda x.Nx) = N \quad (\eta)}{(\lambda x.Mx) = N}\ (\tau)}{M = N}\ (\tau)$$

The second is a very simple derivation of the axiom η in the $\beta\zeta$ calculus.

$$\frac{(\lambda x.Mx)x = Mx \quad (\beta)}{(\lambda x.Mx) = M} \ (\zeta)$$

Hence the $\beta\eta$ and $\beta\zeta$ systems have the same theorems. $\beta\eta \vdash M = N$ if and only if $\beta\zeta \vdash M = N$. $\square$

The ξ rule is essential to proving that $\beta\zeta$ and $\beta\eta$ are equivalent systems, and this is one reason why ξ is known as the principle of weak extensionality. Another reason is that ξ becomes redundant in an extensional system, in the sense that it is provable in the $\beta\zeta$ if we remove the ξ rule. Conversely, ζ is not provable if we remove the ξ rule from the $\beta\eta$ formal system.

Since η gives us an additional reductions $\rhd_\eta$ of the form $(\lambda x.Mx) \rhd_\eta M$, certain β normal forms can be further reduced. The induced reduction operation is still confluent – if $M \rhd M_0$ and $M \rhd M_1$, there is N such that $M_0 \rhd N$ and $M_1 \rhd N$. A term is in $\beta\eta$ normal form if it is in β normal form, and if there are no subterms of the form $(\lambda x.Mx)$, where M does not contain x as a free variable. This is essentially the termination point of computations in the $\beta\eta$ calculus. Since there is more than one $\beta\ \eta$ normal form, even modulo α congruence, the $\beta\eta$ calculus is consistant.

Theorem 4.6. *A term has a β normal form iff it has a $\beta\eta$ normal form.*

Proof. We use the inductive construction of β normal forms to prove that every β normal form has a $\beta\ \eta$ normal form.

- If M is a variable, then M is already in $\beta\eta$ normal form.
- If $M = xN$, and $N \rhd N_0$, where N_0 is in $\beta\eta$ normal form, then $M \rhd xN_0$, and xN_0 is in $\beta\eta$ normal form.
- If $M = \lambda x.N$, and $N \rhd N_0$, where N_0 is in normal form, then $M \rhd \lambda x.N_0$, and either $\lambda x.N_0$ is in $\beta\eta$ normal form, or $N_0 = N_1 x$, where N_1 is in $\beta\eta$ normal form, and $M \rhd \lambda x.N_0 \rhd_\eta N_1$, and so M is reducable to η normal form.

Essentially, η reduction always reduces the complexity of a term, so the induction step works. $\square$

By the same process, we find that if $M \rhd_{\beta\eta} N$, then $M \rhd_\beta M_0$, and $M_0 \rhd_\eta N$, so we can postpone η reduction to the end of calculations.

Combinatory logic is also a non-extensional system. For instance, two prime combinators may have equal axioms without being formally equal in the system. Remember that CL does not satisfy the ξ rule, and thus does not possess the principle of weak extensionality. There is no problem with adding the ξ rule as an axiom, however.

$$\frac{M = N}{[x].M = [x].N}\ (\xi)$$

What's more, we could think of adding the extensionality principles as axioms of CL.

$$\frac{Mx = Nx}{M = N}\ (\xi)$$

$$[x].Mx = M \quad (\eta)$$

where M and N do not contain x as a free variable. The formal theory of equality found by adding the ξ rule is denoted CLξ, and the theory of equality found by adding ζ is denoted CLζ.

Example. *In CL, we find that*

$$SKyx \rhd (Kx)(yx) \rhd x$$

for any combinator y. Therefore in CLζ, we conclude that $SKy = SKz$ for all combinators y and z. Similarily, we find that $KIyx \rhd x$, hence $SKy = KIy$ in CLζ, hence $SK = KI$.

In λ calculus, the ξ rule was essential to proving that $\lambda\beta\eta$ was equivalent to $\lambda\beta\zeta$. This was not so important, because the ξ rule was imbedded in the basic theory. Conversely, in combinatory logic the η rule already holds by definition of the term $[x].Mx$. Using the same techniques as for the extensionality of the λ calculus, we can therefore prove that CLζ is an equivalent theory of equality to CLξ. We remark that we could have left out the special case $[x].Mx$ in the definition of substitution, in which ase the η rule would not hold by definition, and then CLξ would be a strictly weaker theory than CLζ. Various definitions of substitution, and the corresponding ξ rules will result in different systems of equality.

Now to prove the consistency of CLζ, we apply the standard technique: find a system of reduction which specifies the equality, prove confluence, and show that more than one normal form exists. We define **strong reduction** $\triangleright_s$ on combinators to be the standard theory of reduction, with an additional result that if $M \triangleright_s N$, then $[x].M \triangleright_s [x].N$ (and conversely, we sometime denote normal reduction as *weak reduction* $M \triangleright_w N$). I know of no proof of the Church Rosser property for strong reduction which is directly proven from this definition. The main way that strong reduction is proved confluent is by relating the notion directly to the extensional λ calculus. The normal forms here are called **strongly irreducible**. We note, however, that strong reduction is very difficult to work with, which is one reason why little attention to it has been considered.

4.5 Equivalence of Combinatory Formal Systems

Both combinatory logic and the λ calculus accurately model functions obtained by substituting terms. They seem to have very similar properties, albeit from a few small differences. The ξ rule fails in Combinatory logic, whereas the η rule fails in λ calculus. It turns out that if we add these rules to the formal systems, thereby considering their extensional forms, the two systems will have very strong equivalence properties.

First, we shall begin by considering a slight modification to the λ calculus. The terms will not consist of strings of symbols, but rather an equivalence class of strings defined modulo α conversion. In this form of the λ calculus, α reduction does not even need to be considered in the theory, because it is just equality in this modified system. We will let the set of equivalence classes of terms in this calculus be denoted by Λ.

Now given a particular combinatory logic with combinators, with the same variable set as a corresponding λ calculus we shall define a λ **transform** $M \mapsto M_\lambda$ which takes combinators in the logic to α identified terms in Λ which preserves the operation of reduction. This will be the first form of equivalence. If a primitive combinator C has the axiom $Cx_1 \dots x_n \triangleright_C A_C$, then we shall define $C_\lambda = \lambda x_1 \dots x_n.A_C$. We can then define the transform of general combinators by letting $(MN)_\lambda = M_\lambda N_\lambda$. Each M_λ is a **closed** term of the λ calculus, and we call such closed terms combinators, because of this. As a first result, we note that $[M[N/x]]_\lambda = M_\lambda[N_\lambda/x]$.

Lemma 4.7. *If $M \triangleright_w N$ in CL, then $M_\lambda \triangleright_\beta N_\lambda$. Thus if $M =_{CL} N$, then $M_\lambda =_\beta N$. Conversely, if $M =_{CL\zeta} N$, then $M_\lambda =_{\beta\zeta} N_\lambda$.*

This is just proved by induction on the length of a proof of $M \rhd_w N$ and $M =_{\mathrm{CL}\zeta} N$, and is left to the reader.

To obtain an if and only if result, we require a combinatory logic with a basis of combinators expressive enough to represent all possible solutions in the λ calculus. Thus from now on, we assume our combinatory logic contains only the primitive operators S, K, and I. We note that the λ transform of this form of combinatory logic is specified by

$$I_\lambda = (\lambda x.x) \quad K_\lambda = (\lambda xy.x) \quad S_\lambda = (\lambda xyz.xz(yz))$$

We can then define $[x].M$ for any combinatory term M, and this allows us to form an inverse λ transform. We define the CL transform $M \mapsto M_{\mathrm{CL}}$ of any term in Λ. We start by letting $x_{\mathrm{CL}} = x$ for variables x, let $(MN)_{\mathrm{CL}} = M_{\mathrm{CL}}N_{\mathrm{CL}}$, and let $(\lambda x.M)_{\mathrm{CL}} = [x].[M_{\mathrm{CL}}]$. Because $[x].[M_{\mathrm{CL}}]$ does not contain any instances of the variable x, the term is well defined up to α congruence.

Lemma 4.8. *For any combinator M, $[M_\lambda]_{CL} = M$.*

Thus the λ transform has a left inverse. Note, however, that the CL transform is not injective, because

$$(\lambda x.yx)_{\mathrm{CL}} = [x].yx = S(Ky)I$$

and

$$[S(Ky)I]_\lambda = ((\lambda uvw.uw(vw))((\lambda uv.u)y))(\lambda u.u)$$

Though it is surjective.

Lemma 4.9. *If $M =_{\beta\zeta} N$, then $M_{CL} =_{CL\zeta} N_{CL}$.*

As a corollary, we see that $M = N$ in CLζ if and only if $M_\lambda = N_\lambda$ in the $\lambda\beta\zeta$ theory, and $M = N$ in the $\lambda\beta\zeta$ theory if $M_{\mathrm{CL}} = N_{\mathrm{CL}}$ in CLζ.

We note that the correspondence for the reduction rules of CL and λ are nowhere near as elegant. It is easy to prove that if $M \rhd_{\beta\zeta} N$, then $M_{\mathrm{CL}} \rhd_s N_{\mathrm{CL}}$, but the converse does not hold. Since reduction is really only used to form an equivalence of terms, this isn't too much of an issue.

4.6 The Power of λ calculus

The core problem with the λ calculus, and a combinatorially complete combinatory logic, is that it is too expressive in its full form. One result is the following paradox, known as the fixed point theorem.

Theorem 4.10. *For any term M, there is a term N such that $MN =_\lambda N$.*

Proof. Let $N = \lambda x.f(xx)$, and let $M = NN$. Then

$$M = NN = (\lambda x.f(xx))N \rhd_\beta f(NN) = f(M)$$

Thus we have found a fixed point. □

If we are to interpret terms of the λ-calculus as real functions, we must be very careful, because otherwise this theorem would imply every theorem has a fixed point! This is clearly not true for all functions – a particular example in logic is the negation operator, and this theorem would imply $\neg x = x$ for some x. We will address these problems once we have developed a syntax theory for the calculus.

4.7 Models of the λ Calculus and Combinatory Logic

The formal theories of combinatorial logic are fun to play around with, but it is an interesting question what the theory actually *represents*. Combinatory logic is strange among formal systems in the sense that it has no immediate semantic interpretation, because it is immediately too powerful to be represented by set theoretic functions.

Lambda abstraction is a difficult concept to try and model, so let's begin by focusing only on composition, via models of combinatory logic. We cannot model combinators as functions directly, but there is a common method in mathematics to apply elements of sets to the set themselves. For instance, in Hilbert space theory, we can associate with each point x in the space the functional $\langle x|$, which operates on the space by the inner product. In the theory of groups, we can compose elements together by a multiplication operation. Thus we would expect a suitable environment to model combinatory logic as elements of a set X together with a composition operation $X \times X \to X$. The composition operation will be written

$(x,y) \mapsto xy$ except where this becomes ambiguous, and we assume terms are left associative, so we can consider products of the form $x_1 x_2 \ldots x_n$ for $x_i \in X$. Another way of getting around the fact is to represent functions on X as points $x \in X$. We shall let the function associated with $x \in X$ be denoted $f_x : X \to X$. We can then define composition of functions as $(f_x f_y) = f_{f_x(y)}$. This is essentially the same method as with the abstract composition operation, since by currying an association $F : X \to X^X$ of points with functions on sets naturally reduces to $F : X \times X \to X$. We shall call a set X with a composition operation an **applicative structure**.

A function $f : X^n \to X$ is **representable** on an applicative structure X if there is an element $a \in X$, such that $f(x_1, \ldots, x_n) = a x_1 \ldots x_n$ for all $x_i \in X$. Just because we have an applicative structure on a set X, does not imply that *all* functions on the set are representable as elements of the set. In fact, we can guarantee that this does not occur, because Cantor proved that X^X always has a cardinality strictly greater than X, so there cannot exist a surjective map $X \to X^X$. However, we shouldn't expect a model of combinatory logic to model all functions on a set, because one reason for combinatory logic's existence was to model only the *computable functions*.

We shall define a model of combinatory logic with primitive combinators $\mathbf{B}$ to be an applicative structure X with more than one element together with an association $\rho : \mathbf{B} \to X$, where we denote $\rho(C)$ as x_C. If a general combinator C has free variables $y_1, \ldots, y_n$, we will let $x_C[x_1/y_1, \ldots, x_n/y_n]$ be the element of X fromed by mapping y_i to x_i, and then considering the term closed under composition. For a model of combinatory logic to be a true model, the association $x_C x_1 \ldots x_n = A_C[x_1/y_1, \ldots, x_n/y_n]$ for any $x_i \in X$, where C has a substitution axiom $C y_1 \ldots y_n \rhd A_B$. Thus the composition rule of the applicative structure naturally represents the substitution rule for each primitive combinator. Now give a model X, and two terms M and N in combinatory logic, we write $X \models M = N$ if, for all choices of $x_i \in X$, $x_M[x_1/y_1, \ldots, x_n/y_n] = x_N[x_1/y_1, \ldots, x_n/y_n]$, where the y_i are free variables in M and N. This gives us a semantic theory for combinatory logic.

Note that what we have done is not really any more general than the model theory for first order logic, since we can write the terms of combinatory logic as terms of a first order system with a single equality predicate, where the primitive combinators are constants. A model is then just a set X together with a map f from the primitive combinators to X, and we can define an applicative structure on X by taking the interpretation of composed terms in the first order logic.

Two terms in $a, b \in X$ are **extensionally equivalent** if $ax = bx$ for all $x \in X$, or equivalently, if $f_a = f_b$. An applicative structure X is extensional if $f_a = f_b$ holds if and only if $a = b$. A model of extensional combinatory logic should naturally be an extensional applicative structure, and this is certainly true if we interpret a model of this logic as a normal model of the first order theory. By consistency, there exists a model of extensional combinatory logic.

Part II

Computability

In 1931, Kurt Gödel proved all sufficiently complicated axiomatic systems had unprovable theorems, but a fundamental question remained: by what method could we decide whether a theorem could be proved? It took a decade for Alonzo Church and Alan Turing to deduce the impossibility of such a claim. Fifty years later, 'theoretical computation' had become a common reality. In this section, we introduce the mathematical models which formed the foundation for the computer, as well as more modern models which analyze the limitations of various computational methods.

Turing and Church's major breakthrough was precisely defining what a 'computational procedure' is. It is often the case that precise definitions give rise to easy proofs of the most surprising consequence. We shall spend many chapters contemplating what power a computational procedure should have. Philosophically, one should be able to define such a procedure without reference to a computer, for humans computed long before microchips. On the other hand, models should reflect physical reality, since one needs a physical mechanism in order to compute, whether electronic or mental. If your computational model is too strong or too weak, it will not accurately represent the limitations of real life.

We will begin by analyzing the automaton, a model of computation without stored memory. We will expand the amount of expression of the automaton by considering context-free grammars. Finally, we add memory by considering a Turing machine. It is the Church Turing thesis that this is the ultimate model of computation – any real world computation can be modelled as an action on a Turing machine. There has been no evidence in the past century to contradict this thesis, and every realistic model of computation is not as powerful as that of the Turing machine, so we accept the claim. From this model, and with the hypothesis of Church and Turing, we can make precise, philisophically interesting statements about the nature of computation in the real world, addressing theorems about the uncomputability and complexity of certain problems.

As in mathematical logic, the objects of study are strings of symbols over a certain alphabet. One studies the notion of computation syntactically. One of the main ideas of computability theory is that a mental decision can be modelled as a **decision problem** – find a computational model which will 'accept' certain strings over an alphabet. Suppose our problem is to verify whether the addition of two numbers is correct. We are given a, b, and c, and we must decide whether $a + b = c$. Our symbol set is $\{0, 1, \dots, 9, :\}$, and we wish to model a computation which accepts all strings of the form "$a : b : c$", where a, b, and c are decimal strings for which $a + b = c$. Thus we must design machine to accept strings in a specified language, and determining whether a problem is solvable reduces to studying the structure of languages over a finite alphabet. We shall find that, obviously, this problem is possible to compute on a Turing machine, but it is not so simple – there are some models of computation which are unable to decide whether addition is correct.

As a more dynamic discipline than mathematical logic, we need more operations on strings to obtain languages from other languages. We obviously need concatenation, but also **reversal**, which will be denoted s^R. These operations are extended to languages by applying the operations on a component by component basis:

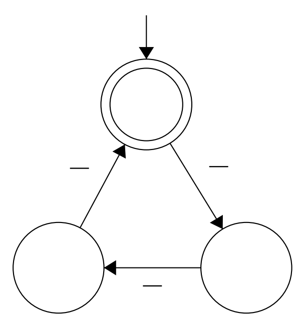
$$S \circ W = \{s \circ w : s \in S, w \in W\} \qquad S^R = \{s^R : s \in S\}$$

A **palindrome** is a string s for which $s^R = s$. If Σ is a set of strings, we shall let $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$.

Chapter 5

Finite State Automata

Our initial model of computability is a computer with severely limited, finite amount of memory. Surprisingly, we shall still be able to compute a great many things. The idea of this model rests on explicitly representing memory as a finite amount of states, whose behaviour is uniquely determined by the state upon which it stands at a point of time. We can represent this process via a state diagram. Suppose we would like to describe an algorithm determining if a number is divisible by three. We shall represent a number by a string of dashes. For instance, "$-----$" represents the number 5. We describe the algorithm in a flow chart below



The algorithm proceeds as follows. Begin at the top node. we proceed clockwise around the triangle, moving one spot for each dash we see. If, at the end of our algorithm, we end up back at the top node, then the number of dashes we have seen is divisible by three. The basic idea of the finite automata is to describe computation via these flow charts – we follow a string around a diagram, and if we end up at a 'accept state', then we accept the string. A mathematical model for this description is a finite state automaton.

A **deterministic finite state automaton** is a 5-tuple $(Q, \Sigma, \Delta, q_0, F)$, where Q is a finite set of states, Σ is a finite alphabet, $q_0 \in Q$ is a start state, $F \subset Q$ are the accept states, and $\Delta : Q \times \Sigma \to Q$ is the transition function. A finite state machine 'works' exactly how our original algorithm worked. Draw a directed graph whose nodes are states, and draw an edge between a state q and each related state $\Delta(q, \sigma)$, for each symbol $\sigma \in \Sigma$. Take a string $s \in \Sigma^*$. We begin at the start state q_0. Sequentially, for each symbol in s, we follow the directed edge from our current state to the state on the edge related to the current symbol in s. If, we end at an accept state in F, then s is an 'accepted' string. Formally, we define this method by extending Δ to $Q \times \Sigma^*$. We define, for $s \in \Sigma^*$, $t \in \Sigma$,

$$\Delta(q, \varepsilon) = q \qquad \Delta(q, st) = \Delta(\Delta(q, s), t)$$

A state machine M **accepts** a string s in Σ^* if $\Delta(s, q_0) \in F$. We call the set of all accepting strings the **language** of M, and denote the set as $L(M)$. A subset of Σ^* which is a language of a deterministic finite state automata is known as a **regular language**.

Example. *Consider $\Sigma = \{-\}$. Then the set of all 'dashes divisible by three' is regular, as in the introductory diagram. Formally, take*

$$Q = \mathbf{Z}_3 \qquad \Delta(x, -) = x + 1 \qquad q_0 = 0 \qquad F = \{0\}$$

then $(Q, \Sigma, \Delta, q_0, F)$ recognizes dashes divisible by three. The 'graph' of the automata is exactly the graph we've already drawn.

Arithmetic is closed under certain operations. Given two numbers, we can add them, subtract them and multiplication, and what results is still a number. In the theory of computation, the operations have a different flavour, but are nonetheless just as important. We shall find that all regular languages can be described from very basic languages under certain compositional operators, under which the set of regular languages is closed.

Theorem 5.1. *If $A, B \subset \Sigma^*$ are regular languages, then $A \cup B$ is regular.*

Proof. let $M = (Q, \Sigma, \Delta, q_0, F)$ and $N = (R, \Sigma, \Gamma, r_0, G)$ be automata recognizing A and B respectively. We shall define a finite automata recognizing $A \cup B$. Define a function $(\Delta \times \Gamma) : (Q \times R) \times \Sigma \to (Q \times R)$, by letting

$$(\Delta \times \Gamma)(q, r, \sigma) = (\Delta(q, \sigma), \Gamma(r, \sigma))$$

Consider

$$H = \{(q, r) \in S : q \in F \text{ or } r \in G\}$$

We contend that

$$((Q \times R), \Sigma, \Delta \times \Gamma, (q_0, r_0), H)$$

recognizes $A \cup B$. By induction, one verifies that for any $s \in \Sigma^*$,

$$(\Delta \times \Gamma)(q, r, s) = (\Delta(q, s), \Gamma(r, s))$$

Thus $(\Delta \times \Gamma)(q_0, r_0, s) \in H$ if and only if $\Delta(q_0, s) \in F$ or $\Gamma(r_0, s) \in G$. □

Theorem 5.2. *If A is a regular language, then A^c is regular.*

Proof. If $M = (Q, \Sigma, \Delta, q_0, F)$ recognizes A. Then define a new machine $N = (Q, \Sigma, \Delta, q_0, F^c)$. The transition $\Delta(q_0, s)$ is in F^c if and only if $\Delta(q_0, s)$ is not in F. □

Corollary 5.3. *If A and B are regular languages, then $A \cap B$ are regular.*

Proof. $A \cap B = (A^c \cup B^c)^c$. □

5.1 Non Deterministic Automata

An important concept in computability theory is the introduction of non-determinism. Deterministic machines must follows a set protocol when understanding input. Non deterministic machines can execute one of many different specified protocols. If any of the protocols accepts the input, then the entire machine accepts the input. Thus non-deterministic machines are said to multitask, for they can be seen to run every protocol specified at once, checking one of a great many protocols to see a pass. An alternative viewpoint is that the machines make a lucky guess – they always seem to choose the write protocol which results in an accepted string.

A **non-deterministic finite state automaton** is a 5-tuple $(Q, \Sigma, \Delta, q_0, F)$, where Q is a finite set of states, Σ is a finite alphabet, q_0 is the start state, $F \subset Q$ are the accept states, and $\Delta : Q \times \Sigma_\varepsilon \to \mathcal{P}(Q)$ is the non-deterministic transition function.

In a non-deterministic finite state automata, we **accept** a string s if $s = s_1 \dots s_n$, where each $s_i \in \Sigma_\varepsilon$, and there are a sequence of states $t_0, \dots, t_{n+1}$, with $q_0 = t_0$ and $t_n \in F$, such that $t_{k+1} \in \Delta(t_k, s_k)$. The set of accepting strings of a machine M form the language $L(M)$. We draw a graph with nodes Q, and with directed edges v to w if $w \in \Delta(v, \Sigma_\varepsilon)$. We begin at q_0. For a string s, we attempt to find a path from q_0 to an accept state, by following edges whose corresponding symbol is in s (or whose symbol is ε, in which we get for free). The string is accepted if such a path is possible. Some call non-deterministic methods a lucky guess methods, since they always make a lucky guess of which deterministic path to take to accept a string.

There is a nicer criterion of acceptance than described above, which is easier to work with in proofs. First, assume there are no ε-transitions in a machine M; that is, $\Delta(q, \varepsilon) = \varnothing$ for all states q. We may then extend $\Delta : Q \times \Sigma \to \mathcal{P}(Q)$ to $\Delta : \mathcal{P}(Q) \times \Sigma^* \to \mathcal{P}(Q)$ recursively by

$$\Delta(X, \varepsilon) = X \qquad \Delta(X, st) = \Delta(\Delta(X, s), t)$$

where $\Delta(X, s) = \{\Delta(x, s) : x \in X\}$. A string s is accepted by M if and only if an accept state is an element of $\Delta(q_0, s)$. If $s = s_1 \dots s_n$, and there are $t_0, \dots, t_n$ with $t_0 = q_0$, t_n an accept state, and $t_{k+1} \in \Delta(t_k, s_k)$, then by induction one verifies that $t_n \in \Delta(q_0, s)$. Conversely, an induction on s verifies that if $q \in \Delta(q_0, s)$, and if $s = s_1 \dots s_n$, then there is a state q' with $q' \in \Delta(q_0, s_1 \dots s_{n-1})$, $q \in \Delta(q', s_n)$. But this implies that if there is an accept state in $\Delta(q_0, s)$, then s is accepted by M. We can always perform this trick, for we may always remove ε transitions.

Lemma 5.4. *Every non-deterministic automata is equivalent to a non deterministic automata without ε transitions, in the sense that they both recognize the same language.*

Proof. Consider a non-deterministic automata.

$$M = (Q, \Sigma, \Delta, q_0, F)$$

Define a state u to be ε reachable from t if there is a sequence of states $q_0, \dots, q_n$ with $q_0 = t$, $q_n = u$, and $q_{i+1} \in \Delta(q_i, \varepsilon)$. Let $E(u)$ be the set of all states ε reachable from u. Define

$$N = (Q, \Sigma, \Delta', q_0, F)$$

Where

$$\Delta'(q,s) = \begin{cases} \bigcup_{u \in E(q)} \Delta(u,s) & s \neq \varepsilon \\ \varnothing & s = \varepsilon \end{cases}$$

Then it is easily checked that $L(N) = L(M)$, for we may skip over ε transitions in N. $\square$

It seems to be a much more complicated procedure to find if a string is accepted by a non-deterministic automata, but it turns out that every non-deterministic automata can be converted into a deterministic automata. The proof relies on the fact that we may exploit the operations of non-determinism, described using power sets of a set.

Theorem 5.5. *A non-deterministic finite state automata language is regular.*

Proof. Let $M = (Q, \Sigma, \Delta, q_0, F)$ be a non-deterministic automata. Assume M has no ε transitions, without loss of generality. Let

$$N = (\mathcal{P}(Q), \Sigma, \Gamma, \{q_0\}, \{S \in \mathcal{P}(Q) : S \cap F \neq \varnothing\}$$

where

$$\Gamma(S,t) = \Delta(S,t)$$

We have already verified this deterministic machine recognizes $L(M)$, for $\Delta(\{q_0\}, s)$ contains an accept state if and only if s is accepted. $\square$

It is now fair game to use non-deterministic automata to understand regular languages, for the language of every non-deterministic automata is regular.

Theorem 5.6. *If A and B are regular languages, then $A \circ B$ is regular.*

Proof. Let $M = (Q, \Sigma, \Delta, q_0, F)$ and $N = (R, \Sigma, \Gamma, r_0, G)$ be deterministic automata accepting A and B respectively. Without loss of generality, assume Q is disjoint from R. Consider the non-deterministic machine

$$O = (Q \cup R, \Sigma, \Pi, q_0, G)$$

Define

$$\Pi(s,t) = \begin{cases} \{\Delta(s,t)\} & s \in Q, \text{and } t \neq \varepsilon \text{ or } s \notin F \\ \{\Delta(s,t), r_0\} & s \in F, t = \varepsilon \\ \{\Gamma(s,t)\} & s \in R \\ \varnothing & \text{otherwise} \end{cases}$$

We quickly verify that if $s \in L(M)$ and $w \in L(N)$, then $sw \in L(O)$. If $v \in L(O)$, there is some substring k such that $r_0 \in \Pi(q_0, k)$ (for this is the only path from q_0 to G). If we choose k to be the shortest such string, then k must also be an accept string in $L(M)$, since any substring of k cannot map to states in R, and thus k must map via the ε transition from an accept state of M. If $v = kr$, then $\Pi(r_0, r)$ is an accept state in N, so $r \in L(N)$. Thus $v \in L(M) \circ L(N)$. □

Theorem 5.7. *If A is a regular language, A^* is regular.*

Proof. Let $M = (Q, \Sigma, \Delta, q_0, F)$ be a deterministic language which accepts A. Form a non-deterministic automata $N = (Q \cup \{i\}, \Sigma, \Gamma, i, F)$, where

$$\Gamma(q,s) = \begin{cases} \{\Delta(q,s)\} & s \neq \varepsilon \\ \{q_0\} & s = \varepsilon, q = i \\ \{i\} & s = \varepsilon, q \in F \\ \varnothing & \text{otherwise} \end{cases}$$

Then $L(N) = A^*$, for if $s = a_1 \dots a_n$, with $a_i \in A$, then we may go from i to q_0, then to an accept state via a_1, return to i with a ε transition, and continue, so s is accepted. If we split a string accepted to $L(N)$ into when ε transitions to i are used, then we obtain strings in A. □

In turns out that the operations of concatenation, union, and 'kleene starification' are enough to describe all regular languages. Thus we may take an algebraic approach to understanding regular languages, using the symbology of regular expressions.

5.2 Regular Expressions

Automata are equivalent to a much more classical notion of computation – regular expressions.

Definition. A **regular expression** over an alphabet Σ is the smallest subset of $(\Sigma \cup \{\emptyset, \cup, *, (,)\})^*$ such that

1. $\emptyset$ is a regular expression, as are $s \in \Sigma^*$.
2. If λ and γ are regular expressions, then so are $\lambda^*, (\lambda \cup \gamma)$, and $(\lambda \circ \gamma)$.

Every regular expression describes a language. A string $s \in \Sigma^*$ is recognized by an regular expression λ if

1. $\lambda \in \Sigma^*$, and $s = \lambda$.
2. $\lambda = (\lambda \cup \gamma)$, and s is recognized by λ or by γ.
3. $\lambda = \gamma \circ \delta$, and $s = wl$, where w is recognized by γ, and δ recognizes l.
4. $\lambda = \gamma^*$, and $s = \varepsilon$ or $s = s_1 \dots s_n$, where s_i is recognized by γ.

The regular language corresponding to a regular expression λ is $L(\lambda)$, the set of strings recognized by t. The set of all regular expressions on an alphabet Σ will be denoted $R(\Sigma)$.

It is a simple consequence of our discourse that every language recognized by a regular expression *is actually* a regular language. In fact, we can show that every regular language is described by a regular expression. We shall describe an algorithm for converting a finite state automaton to a regular expression. We shall make a temporary generalization, by allowing non deterministic finite automata to have regular expressions in their transition functions. A **generalized non-deterministic finite state automaton** is a 5-tuple $M = (Q, \Sigma, \Delta, q_0, f_0)$, where $\Delta : Q - \{f_0\} \times Q - \{q_0\} \to R(\Sigma)$ is the generalized transition function, and q_0 is the start state, f_0 is the end state. A generalized automaton accepts $s \in \Sigma^*$ if we may write $s = s_1 \dots s_n$, and there are a sequence of states $q_0, \dots, q_n$ where $q_n = f_0$, and $\Delta(q_i, q_{i+1})$ recognizes s_i.

Theorem 5.8. *Any generalized finite-state automaton describes the language of a regular expression.*

Proof. If the generalized automaton has two states, then there is only one transition from start state to begin state, and this transition describes a regular expression for the automaton. We will reduce every automaton to this form by induction. Suppose an automaton has n states. Fix a state $q \in Q$, which is neither the beginning or accepting state. Define a new automaton

$$N = (Q - \{q\}, \Sigma, \Delta', q_0, f_0)$$

where $\Delta'(a,b) = (\Delta(a,b) \cup \Delta(a,q)\Delta(q,q)^*\Delta(q,b))$ Then N is equivalent to M, and has one fewer state, and is thus equivalent to a regular expression. □

Corollary 5.9. *Every regular language is described by a regular expression.*

Proof. Clearly, every DFA and NFA is equivalent to a generalized NFA, for by adding new states, we may ensure no states map back to the start state, and that there is only one end state. □

5.3 Limitations of Finite Automata

We've discovered a menagerie of different problems we can solve with finite automata, but it has already been foreshadowed that better machines await. Here we discover methods which attack languages, showing that they cannot be recognized by regular expressions, not finite automata.

Theorem 5.10 (Pumping Lemma)**.** *Let L be a regular language. Then there is a number p, called the pumping length, such that any $s \in L$ which satisfies $|s| \geqslant p$, then we may write $s = wuv$, where $|u| > 0$, $|wu| \leqslant p$, and $wu^iv \in L$ for all $i \geqslant 0$.*

Proof. Let L be a regular language, and M a deterministic automata recognizing L with p states. Let s be a string with $|s| \geqslant p$, with $s \in L(M)$. Write $s = s_1 \dots s_n$ and let $q_k = \Delta(q_0, s_1, \dots, s_k)$. Then we obtain $|s| + 1$ states $q_0, q_1, \dots, q_{|s|}$. By the pidgeonhole principle, since q_i equals some q_j, for $i < j$. Let $w = s_1 \dots s_i$, $u = s_{i+1} \dots s_j$, $v = s_{j+1} \dots s_{|s|}$. Then $\Delta(\Delta(q_0, w), u) = \Delta(q_0, w)$, so

$$\Delta(q_0, wu^iv) = \Delta(q_0, wuv)$$

So $wu^iv \in L$ for all i. □

Example. *$L = \{0^k 1 0^k : k \geqslant 0\}$ is not regular. If it was regular, it would have a pumping length p. Since $0^p 1 0^p$ is in L, so we may write $0^p 1 0^p = wuv$, where $|wu| \leqslant p$, and $wu^i v \in L$. Then $u = 0^k$, for $k > 0$, and $wv = 0^{p-k} 1 0^p \in L$, which is clearly not in the language, contradicting regularity.*

Example. *$L = \{1^{n^2} : n \in \mathbf{N}\}$ is not regular. Suppose we had a pumping length p. Then $1^{p^2} \in L$, so there is $0 < k \leqslant p$ with $1^{p^2+k} \in L$. But*

$$(p+1)^2 = p^2 + 2p + 1 > p^2 + k$$

] And there is no perfect square between p^2 and $(p+1)^2$, a contradiction.

Example. *The language $L = \{0^i 1^j : i > j\}$ is not regular. If we had a pumping lenght p, then $0^p 1^{p-1} \in L$. But then there is $0 < k \leqslant p$ such that $0^{p+(i-1)k} 1^{p-1} \in L$ for all natural numbers i. In particular, for $i = 0$, we find $0^{p-k} 1^{p-1} \in L$ But $p - k \leqslant p - 1$, a contradiction.*

There is a much more mathematically elegant and complete way of separating regular languages from non-regular ones, discovered by John Myhill and Anil Nerode. Consider an alphabet Σ, and a particular language $L \subset \Sigma^*$. Call two strings a and b in Σ^* L-indistinguishable if $az \in L$ if and only if $bz \in L$ for any $z \in \Sigma^*$. This forms an equivalence relation on Σ^*. We shall define the index of L, denoted Ind L, to be the cardinality of the partition.

Theorem 5.11 (Myhill-Nerode)**.** *L is regular if and only if Ind $L < \infty$, and Ind L is the number of states of the smallest finite state machine to recognize L.*

Proof. Let M be a deterministic finite state machine recognizing L with n states, transition function Δ, and start state q_0. Let $a_1, \dots, a_{n+1}$ be $n+1$ strings in Σ^*. We claim that at least one pair is indistinguishable. if we take $q_i = \Delta(q_0, a_i)$, then some $q_i = q_j$. These two strings are then indistinguishable in L. Conversely, suppose that Ind $L < \infty$. Let $A_1, \dots, A_n$ be the equivalence classes of Σ^*. If $s \in A_i$ is contained in L, then every other $w \in A_i$ is in L, for otherwise s and w can be distinguished. Build a finite state machine whose states are A_i, whose start state is $[\varepsilon]$, and whose transition function is

$$\Delta([s], t) = [st]$$

$[s]$ is accepted if and only if $s \in L$. This finite state machine recognizes s. ∎

In some circumstances, the Myhill Nerode theorem is very powerful.

Example. *For $\Sigma = \{a, b, \dots, z\}$, consider the set L of words w whose last letter has not appeared before. For example, the words "apple", "google", "k", and ε are in L, but the words "potato" and "nutrition" are not. Is this language regular? We apply Myhill Nerode. If the letters in one word are different to the letters in another word, these words are distinguishable. If the letters in one word are the same as the letters in another word, and both are not accepted or both are not accepted, these words are indistinguishable. Thus the index of the language in consideration is the same as the number of different subsets of the set of letters in a word, counted twice for repeated and non-repeated characters. Subtracting one from the fact that ε need only be counted once, we find that* $2 \cdot 2^{26} - 1 = 2^{27} - 1$. *Since this is finite, the language is regular, and this is the minimal number of sets in a finite state machine recognizing the language.*

5.4 Function representation

Decision problems are sufficient to model a large variety of computational models, but for completeness, we should at least mention how we determine whether problems with multiple outputs can be. The counterpart of a finite-state automata is a finite-state transducer. A **finite state transducer** is a 5 tuple $(Q, \Sigma, \Lambda, \Delta, \Gamma, q_0)$, where Q is the set of states, Σ is the input alphabet, Λ is the output alphabet, $\Delta : Q \times \Sigma \to Q$ is the transition function, $\Gamma : Q \times \Sigma \to \Lambda_\varepsilon$ is the transduction function, and q_0 is the start state. Each finite-state transducer M gives rise to a function $f_M : \Sigma^* \to \Lambda^*$, defined by

$$f_M(\varepsilon) = \varepsilon \qquad f_M(st) = f_M(s) \circ \Gamma(\Delta(q_0, s), t)$$

A **regular function** is one computable by a finite state transducer.

Example. *Consider an alphabet* $\mathbf{F}_2$ *and consider the function f taking and returning strings over* $\mathbf{F}_2$*, inverting the strings on even positions, and leaving the strings on the odd positions. Then f is a regular function, for it may be computed by the automata below.*

Chapter 6

Context Free Languages

Finite state machines are useful on a specific set of problems, but have limitations. We would like our notion of computability to decide on a much more complicated set of problems. Thus we must define new classes of computable languages.

6.1 Context Free Grammars

Definition. A **Context Free Grammar** is (V, Σ, R, S), where V is a set of variables, Σ is a character set, disjoint from V, R is a relation between V and $(V \cup \Sigma)^*$, and $S \in V$ is the start variable. We call elements of R derivation rules, and write $(a, s) \in R$ as $a \to s$.

A string of the form uvw is **directly derivable** from uAw if $A \to v$ is a derivation rule. The smallest transitive relation containing the 'directly derivable' relation is the derivability relation. To reiterate, a string u is **derivable** from w if there is a sequence of direct derivations

$$w \to s_0 \to \cdots \to s_n \to u$$

Such a sequence is known as a **derivation**. The language of a grammar is the set of all strings in Σ^* derivable from the start state S. As in finite automata, the language of a grammar G will be denoted $L(G)$. A language is **context-free** if it is the language of a context free grammar.

Lemma 6.1. *Let $G = (V, \Sigma, R, S)$ and $H = (V, \Sigma, R', S)$ be two different context free languages. If every $(v, s) \in R'$ is a derivation in G, then $L(H) \subset L(G)$.*

Proof. If we can show that every direct derivation in H is a derivation in G, then all derivations in H are derivations in G, since derivations form the smallest transitive relation containing the direct derivations in H, and the derivations in G certainly satisfy this. If w is directly derived from s in H, then there is a rule (B, u), $s = rBt$, and $w = rut$. There is a sequence $s_0 \dots s_n$ deriving u from B in G. But then $(rs_0t)\dots(rs_nt)$ is a derivation of w from s, so w is derivable from s in G. □

A leftmost derivation is a derivation which always swaps out the leftmost variable. A string is **ambiguous** if it has two different leftmost derivations. A grammar is ambiguous if its language contains an ambiguous string. Ambiguity is unfortunate when parsing a language, since it means we may be able to interpret elements of the language in two different ways.

Example. *First order logic can be defined as an unambiguous grammar. To develop the language, we took a bottom up approach, but a top up approach can also be taken. We must take a finite alphabet to develop the language, which we take to be*

$$\{(,), \forall, \exists, \neg, \wedge, \vee, \Rightarrow, x, f, P, 0, 1, \dots, 9\}$$

we cannot take an 'infinite number of variables', in the sense of an infinite number of symbols, for then formal language theory does not apply. We must instead assume our variables, predicates, and functions are themselves words in a finite alphabet. For instance, we will enumerate our variables

$$\Lambda = \{x, x_0, x_{00}, \dots, x_{0000000000}, \dots\}$$

The trick to forming functions and predicates is to put n ones after an f or a P to denote that is is an n-ary function, so

$$\mathcal{F}^n = \{f^{111\dots11}, f^{111\dots11}{}_0, \dots, f^{111\dots11}{}_{000000000}, \dots\}$$

$$\mathcal{P}^n = \{P^{1111\dots11}, \dots\}$$

Then we may form predicate logic as a context free grammar. The variables are easiest to form

$$X \to x \mid X_0$$

Terms are tricky to define because we must form functions as well. The trick is to introduce new variables Y and Z which add enough terms to the function.

$$\begin{aligned}
T &\to X \mid f^1 U) \\
U &\to V(T \mid {}^1 U, T \\
V &\to \varepsilon \mid V_0
\end{aligned}$$

Finally, we form the formulas of the calculus. Again, the only trick part are the atomic formulae

$$\begin{aligned}
F &\to (F \wedge F) \mid (F \vee F) \mid (\neg F) \mid (F \Rightarrow F) \mid (\forall x : F) \mid (\exists x : F) \mid P^1 Z) \\
Y &\to Z(T \mid {}^1 Y, T \\
Z &\to \varepsilon \mid Z_0
\end{aligned}$$

We showed the formulas are derived unambiguously, but this took a lot of hard work. It is impossible to find a general procedure to decide whether a language is ambiguous, which is what makes verifying ambiguity so difficult.

It is useful to put grammars in a simple form for advanced theorems. A grammar (V, Σ, R, S) is in **Chomsky Normal Form** if the only relations in R are of the form

$$S \to \varepsilon \qquad A \to BC \qquad A \to a$$

where $A, B, C \in V$ and $B, C \neq S$, and $a \in \Sigma$.

Theorem 6.2. *Every context-free language can be recognized by a context-free grammar in Chomsky normal form.*

Proof. We shall reduce any context free grammar $G = (V, \Sigma, R, S)$ to a context free grammar in normal form in a systematic fashion, adding each restriction one at a time.

1. **No derivation rules map to the start variable**:
   Create a new start variable mapping onto the old start variable.

2. **There are no ε-transition rules except from the start variable**:
   Define a variable $A \in V$ to be nullable if we may derive ε from A. Let W be the set of all nullable variables. Define a new language $G'(V, \Sigma, R', S)$ such that, if $A \to A_1 \dots A_n$ is a derivation rule in G, and

$A_{i_1}, \dots, A_{i_m}$ are nullable, then we add 2^m new rules to G' by removing some subset of the A_{i_k}. Then $L(G') = L(G) - \{\varepsilon\}$, so that if $\varepsilon \in L(G)$, we need only add an ε rule to S to make the two languages equal.

We will prove that if A is a variable in G', then A can derive $w \in \Sigma^*$ in G' if and only if it can derive it in G and $w \neq \varepsilon$. One way is trivial, the other a proof by induction on the length of the derivation. Suppose we have a derivation in G

$$A \to s_0 \to \cdots \to s_n \to w$$

Let $s_0 = A_1 \dots A_n$. Then each A_i derives w_i in G, where $w = w_1 \dots w_n$. We can choose such a derivation to be shorter than the derivation of G. But this implies that A_i derives w_i in G', provided $w_i \neq \varepsilon$. Let $w_{i_1} \dots w_{i_m} \neq \varepsilon$. Then $m \neq 0$, since $w \neq \varepsilon$. We have a corresponding production rule $A \to A_{i_1} \dots A_{i_n}$ in G', since the other variables are nullable. Thus, by induction, A can derive w.

3. **There are no derivation rules $A \to B$, where B is a variable**:
   Call B unitarily derivable from A if there are a sequence of derivation rules
   $$A \to V_1 \to \cdots \to V_n \to B$$
   Define a new grammar $G' = (V, \Sigma, R', S)$. If B is directly derivable from A, and B has a derivation rule $B \to s$, then G' has a production rule $A \to s$, provided that s is not a variable. Then G' has no rules of the form $A \to B$, and generates the same language. This is fairly clear, and left to the reader to prove.

4. **Every rule is of the form $A \to AB$ or $A \to a$**:
   If we have a rule in G of the form $A \to s$, where $s = s_1 \dots s_n$, and $s_{i_1}, \dots, s_{i_m} \in \Sigma$, then add new unique variables $A_{s_{i_k}}$ for each $s_i \in \Sigma$, and replace the rule with new rules of the form
   $$A \to s_1 \dots A_{s_{i_1}} \dots A_{s_{i_m}} \dots s_n$$
   $$A_{s_{i_m}} \to s_{i_m}$$
   Thus we may assume every rule of the form $A \to A_1 \dots A_n$ (where we may assume $n \geqslant 2$) only maps to new variables. But then we may add new variables $V_2 \dots V_{n-1}$, and swap this rule with rules of the form
   $$A \to V_{n-1} A_n$$

$$V_k \to V_{k-1}A_k$$
$$V_2 \to A_1A_2$$

Now every derivation rule is in the correct form, and we have reduced every grammar to Chomsky normal form. $\square$

Chomsky normal form allows us to prove a CFG pumping lemma.

Theorem 6.3. *If L is a context free language, then there is $p > 0$, such that if $s \in L$, and $|s| \geqslant p$, then we may write $s = uvwxy$, where $|vwx| \leqslant p$, $v, x \neq \varepsilon$, and for all $i \geqslant 0$, $uv^iwx^iy \in L$.*

Proof. Let A be the language of the grammar G, which we assume to be in Chomsky normal form. Let there be v variables in G. If the parse tree of $s \in A$ has height k, then $|s| \leqslant 2^{k-1}$, which follows because the tree branches in two except at roots, so there is at most 2^{k-1} roots. If $|s| \geqslant 2^{3v}$, then every parse tree of s has height greater than v. Pick a particular parse tree of smallest size. There is a sequence of variables $A_1, A_2, \dots, A_{3v}$, such that A_i is the parent of A_{i+1}. Because of how many variables there are, some variable must occur at least 3 times (for otherwise we may remove the variables in pairs, to conclude that $3v - 2v = v$ variables contain no variables, a contradiction). What's more, they much occur within a height of $3v$ of each other. Let $A_i = A_j = A_k$, for $i < j < k$. Let A_i produce aA_jb, let A_j produce xA_ky, and let A_k produce r. Write $s = maxrybn$. By virtue of the minimality of the tree, we may assume that a or b is nonempty, and one of x or y is nonempty. First, if both ax and yb are assumed non-empty, then we may pump these strings up, and have proved our lemma. So suppose ax is empty. Then b and y are nonempty, and $s = mrybn$. Since A_i produces A_jb, and A_j produces A_ky, A_i may be pumped to produce A_jb^i, A_j may be pumped to produce A_ky^i, and mry^ib^in is in the context free language, so we have non-empty strings to pump. The proof is simlar if by is empty, for then a and x are non-empty. The constraint $|vxy| \leqslant k$ is satisfies for yb and ax, for the A_i and A_j lie within $3v$ of each, so the string produces in this production is at most as long as 2^{3v}, which is less than or equal to the pumping length. $\square$

6.2 Pushdown Automata

Regular languages have representations as the languages of regular expressions or as finite automata. Context-free languages also have dual representations, as 'machines' or as abstraction operations. It is good to represent a language as a machine for it may hint as to what hardware capabilities a computer must have to be able to solve problems related to the languages. The key machine component for a context-free language is a stack. A pushdown automata is a finite state automata with the addition of a stack.

Definition. A **(non-deterministic) pushdown automata** is a tuple $(Q, \Sigma, \Lambda, \Delta, q_0, F)$, where Q is a finite set of states, Σ is an alphabet, Λ is the stack alphabet, $\Delta : Q \times \Sigma_\varepsilon \times \Lambda_\varepsilon \to \mathcal{P}(Q \times \Lambda_\varepsilon)$ is the state transition function, $q_0 \in Q$ is the start state, and $F \subset Q$ are the accept states.

It turns out that deterministic pushdown automata are less powerful than non-deterministic automata, so we do not discuss deterministic automata. It is interesting to note that the languages of deterministic automata are connected to unambiguous grammars, though we will not have time to discuss this further.

Let us describe a pushdown automata intuitively. The automata has a stack of symbols from Λ, which it can push and pull from when deciding how to move through the machine. A stack is a string in Λ^*. Thus, formally, a string s is **accepted** by a push-down automata M if there are a sequence of states $q_0, \dots, q_n$, and stacks $w_0, \dots, w_n \in \Lambda^*$ such that $q_n \in F$, $w_0 = \varepsilon$, and if we write $w_i = w\lambda$, with $\lambda \in \Lambda_\varepsilon$, then $w_{i+1} = w_i\lambda'$, with $(q_{i+1}, \lambda') \in \Delta(q_i, \lambda)$. Thus we pop and pull off the rightmost character in the string when moving between states.

Pushdown automata have enough versatile memory to recognize context free languages. The stack can 'remember' variables it has yet to parse, and check when symbols are used. We shall allow a mild generalization of pushdown automata, which can push multiple symbols to the stack at a time. This is fine, without loss of generality, because we could have instead introduced new states that take nothing from the stack, and push the symbols on one at a time. We shall also assume a pushdown automata starts with a \$ symbol at the bottom of its stack, which is fine, because we could have added another start state to the automata which pushes the \$ on as we begin running the machine.

Theorem 6.4. *Every context free language is accepted by a pushdown automata.*

Proof. Consider a context free language (V, Σ, R, S). Consider a pushdown automata $(Q, \Sigma, \Lambda, \Delta, q_0, F)$, with the stack language $\Lambda = \Sigma \cup V$, and Q just two states q_0 and f_0. For each derivation $A \to s \in R$ we have

$$(q_0, s) \in \Delta(q_0, \varepsilon, A)$$

And for each $a \in \Sigma$, we have

$$(q_0, \varepsilon) \in \Delta(q_0, a, a)$$

And a finale transition

$$(f_0, \varepsilon) \in \Delta(q_0, \varepsilon, \$)$$

It is clear that this automata parses the context free language. □

A converse also holds, so that pushdown automata are equivalent computers of context free languages. To do this, we assume that the automata pushes everything off its stack before it finishes, and has a single accept state f_0. In addition, we shall assume that a state only pops and pulls in one action, and doesn't do both at the same time. Adding additional states means this is no loss of generality.

Theorem 6.5. *Each pushdown automata language is context free.*

Proof. The gist of our approach is as follows. Let $(Q, \Sigma, \Gamma, \delta, q_0, \{f_0\})$ be a pushdown automata. We shall define a context grammar with variables A_{pq}, with $p, q \in Q$. This variable should be able to generate all possible strings which can start in p with an empty stack, and end up in q with an empty stack. Our start variable will then be A_{q_0, f_0}. The first rules are most basic

$$A_{pp} \to \varepsilon$$

Such a path may end up empty halfway through the path, so we have these rules, for each $p, q, r \in Q$,

$$A_{pq} = A_{pr} A_{rq}$$

We can also pop something on the stack, and save it for a long time later. If $t \in \Sigma$, and $(r,t) \in \Delta(p,a,\varepsilon)$, and $(q,\varepsilon) \in \Delta(s,b,t)$, then we add the derivation rule

$$A_{pq} = aA_{r,s}b$$

We claim these rules describe all possible derivations we could make in the pushdown automata. It is clear that all such derivations in this context language are accepted in the pushdown automata.

We shall prove that if we can move from p to q using a string x, both with an empty stack, then $A_{pq} \to x$. This is done by induction on the number of steps to accept the string in the automata. If we do this in one step, then the string is empty, and we have a rule $(q,\varepsilon) \in \Delta(p,\varepsilon)$, or the string consists of a single letter, and we have a rule $(q,\varepsilon) \in \Delta(p,t)$. In the first case, we have a derivation $A_{p,q} \to \varepsilon$, and in the second, we have a derivation

$$A_{p,q} \to tA_{q,q}\varepsilon \to t\varepsilon\varepsilon = t$$

Now consider a machine that runs for a length n. Suppose the stack empties at some state k, after running through x_1 of the string, for $x = x_1x_2$. Then we have, by induction, a derivation

$$A_{pq} \to A_{pk}A_{kq} \to x_1x_2$$

Thus we may assume that the stack never empties except at beginning and end. Then the first action must be to push a symbol t to the stack, and the last action to remove t. If we move from p to r in the first action by reading a, and from s to q in the last action by reading b, then we may write $x = acb$, and by induction, we have the derivation

$$A_{pq} \to aA_{rs}b \to acb = x$$

Thus we have verified the equivalence of the pushdown automata and context free language. $\square$

Pushdown automata are easy to connect to their finite state cousins.

Corollary 6.6. *Every regular language is context free.*

Chapter 7

Turing Machines and Uncomputability

Finite state machines are good at modelling machines with a small amount of memory, and pushdown automata are good at modelling stack processes. Nonetheless, there are many problems that 'should be computable', since we can solve them which these automata cannot solve. In this chapter, we introduce a much more robust model, the **Turing machine**, which satisfies this criteria. Every known algorithm can be implemented in this model.

The basic idea is that a Turing machine runs off of an infinite tape, which the machine can scan through, swerving left and right to read off the tape. The tape begins with the input

$$q_0 x_1 x_2 \dots x_n$$

A notation which implies that the machine is in state q_0, and the tape head is looking at x_1, and the tape consists of the letters $x_1, x_2 \dots, x_n$, and then the rest of the tape consists of blank spaces.

The Turing machine has finitely many states, which describe how the machine reacts when it sees a current character – it chooses to swap the current character with a different character, and moving either left or right one space. We may also choose to accept the string or reject the string at any time.

Formally, a turing machine can be described as a tuple

$$(Q, \Sigma, \Gamma, \Delta, q_0, q_{\text{accept}}, q_{\text{reject}})$$

where Q is a set of states, Σ is an input alphabet, Γ is a tape alphabet (which we assume includes the blank space $-$), $q_0 \in Q$ is the start state, $q_{\text{accept}} \in Q$ is the accept state, and $q_{\text{reject}} \in Q$ is the reject state, and $\Delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$. We interpret this as given a state and a currently read letter, we move to a new state, we replace the letter with a new character, and we move left or right.

To compute the operation of a Turing machine on a string, we recursively define a computation on a string. A specific state of the machine will be represented by a string sqw, where $q \in Q$, $s, w \in \Gamma^*$. We say that $sqtw$ **yields** $suq'w$ if $(q', u, R) = \Delta(q, t)$.

Chapter 8

Complexity Theory

We have now discovered what it means to solve a problem algorithmically. We formalize the problem into a certain language over an alphabet, then find a Turing machine over that alphabet which terminates on every input, and accepts exactly theres strings that are an element of a language. But just because a problem is solvable, does not mean that it is *feasibly solvable*; there may be an algorithm which will eventually solve a problem, but if it takes centuries to find the answer, the solution is not effective. In this section, we restrict our attention to the computable problems, and describe the degree of difficulty of certain problems. First, we must find a measure of a problem's difficulty, so that we can classify problems based on how difficult they are.

8.1 Measuring Complexity

Let M be a turing machine over an alphabet Σ, which halts on all inputs. Then, for each string $s \in \Sigma^*$, M deterministically executes a certain number of steps, eventually terminating. Define $t_M(s) \in \mathbf{N}$ to be the number of configurations M steps through before it halts on input s. The **time complexity** of M is the function

$$f_M(n) = \max\{t_M(s) : s \in \Sigma^*, |s| = n\}$$

It could be argued that a better notion of complexity is the average running time over inputs of a certain length, but we rarely know how often certain inputs will occur. Often, in practice, slow inputs occur much more often that fast inputs, so worst time analysis better reflects an algorithm's speed. Furthermore, worst time analysis gives us an elegant and useful theory which gives meaningful results, which without this simplification would be impossible. We should not be dogmatic in this approach, because some algorithms do benefit from an average case analysis (for instance, the ellipsoid algorithm in linear programming), but the worst-case approach has turned out to be the most useful.

We rarely specify a turing machine exactly, and thus have difficulty expressing f_M as an exact number. Even if we described a machine exactly, f_M likely will not have a simple formula specifying an algorithms speed. Thus we apply the theory of asymptotics. Recall that a real-valued function f, the 'big O' set $O(f)$ consists of all functions g for which $|g| \leqslant c|f|$ *eventually* holds, where c is some positive constant. Similarily, the 'little o' set $o(f)$ consist of all functions g satisfying the sharper relation

$$\lim_{x \to \infty} \left| \frac{g(x)}{f(x)} \right| = 0$$

Less important to our needs, but canonically include are the $\Omega(f)$ and $\omega(f)$ sets, which consist of functions g such that $f \in O(g)$ and $f \in o(g)$ respectively. We define $\Theta(f) = \Omega(f) \cap O(f)$, the class of functions which are asymptotically equal to f.

Example. *Let $P \in \mathbf{R}[X]$ be a polynomial of degree m. For $x \geqslant 1$, and $i \leqslant j$, the inequality $x^i \leqslant x^j$ holds, so if P is expressed in coefficients as*

$$P = \sum c_i X^i$$

then for $x \geqslant 1$,

$$|P(x)| \leqslant \sum |c_i| x^i \leqslant \left(\sum |c_i| \right) x^m$$

and we have shown that $P \in O(x^m)$.

Example. *Since $O(f)$ is a subset of a space of functions, they can be manipulated under real-valued operations. As an example, we consider the set $2^{O(\log n)}$, which consists of all functions of the form 2^f, where $f \in O(\log n)$. We contend this is just the set of all functions g in $O(n^c)$ for some c. First note that if*

$$f(x) \leqslant c \log x$$

then because the exponential function is monotone, we obtain that

$$2^{f(x)} \leqslant 2^{c\log x} = x^c$$

so if the first inequality eventually holds, the second inequality eventually holds. Conversely, suppose $f \in O(x^c)$. *Suppose that eventually*

$$f(x) \leqslant Kx^c$$

Then if $x \geqslant e$,

$$\log_2 f(x) \leqslant \log_2 K + c\log_2 x \leqslant \frac{c + 2\log_2 K}{\log 2}\log x$$

which implies $\log_2 f \in O(\log n)$, *and* $f = 2^{\log_2 f} \in 2^{O(\log n)}$.

Example. *A commonly used relation is that* $O(f) + O(g) = O(f+g)$. *Let* h *and* k *be functions with a constant* K *and* n_0, *such that*

$$h \leqslant Kf \qquad k \leqslant Kg$$

eventually holds. Then

$$h + k \leqslant K(f+g)$$

eventually holds as well, so $h+k \in O(f+g)$. *Conversely, suppose* $h \in O(f+g)$, *and let*

$$h \leqslant K(f+g)$$

Define

$$h_1 = \min(h, Kf) \qquad h_2 = h - h_1$$

Then $h_1 + h_2 = h$. *It is easy to see* $h_1 \in O(f)$. *Now if* $n \geqslant n_0$, *then*

$$h(n) - h_1(n) \leqslant Kf + Kg - h_1(n) \leqslant Kg$$

Thus $h_2 \in O(g)$, *and we have verified the equality. Similar arguments show*

$$O(fg) = O(f)O(g)$$

Certainly, a machine M with $f_M(n) = n^2$ runs much faster than a machine N with $f_N(n) = 2^8n^2$, and certainly a machine L with $f_L(n) = 2^n$ runs faster than f_N for the first 20 input sizes, but regardless of the constant, the exponential machine will quickly become much slower even if we choose a much larger constant ($2^{100}n^2$ becomes smaller than 2^n only after the input size increases to 125). What's more, we have a theorem that, by constructing a new machine, we can always decrease the constant in an algorithm, with at most a linear increase in time.

Theorem 8.1. *For any two-tape Turing machine M, and $\varepsilon > 0$, there is an equivalent two-tape machine N such that*

$$f_N(x) \leqslant \varepsilon f_M(x) + (1+\varepsilon)n$$

eventually holds.

Proof. Consider the following strategy. Let M have tape alphabet Σ and tape alphabet $\Sigma \cup \Sigma^n$. Construct N with tape alphabet $\Sigma \cup \Sigma^m$, and with states $Q \times \Sigma^m \cup Q'$. Our machine takes the first m characters $w_1, \dots, w_m$ in input, and places $(w_1, w_2, \dots, w_m) \in \Sigma^m$ on the second tape. This takes m steps. If the input does not contain m strings, we assume that we write in whitespace instead. Continue this until we reach whitespace. After $m\lceil n/m \rceil$ steps, we have a condensed input. After this preprocessing, we can execute m steps of the original machine in 6 steps of our algorithm. Our state in N will now be of the form (q,t), where q is a state in M, and t is the current position in the tuple $(w_1, \dots, w_m)$ we are pointing at in the second string that the simulated machine should be at. We move left once, and then right twice, in order to see the states to the left and right of our current states. Given this information, we can predict whether we will end up on the left or right after m steps of the simulated machines, and what the other states will look like. Only states in the current tuple, and the left or right tuple (but not both) will be changed. Based on our computation, we move left or right, changing the states we are currently in, and, if needed, move again to change the states in our new tuple. Thus in six steps, we have simulated m steps. If M takes k steps to halt on some input, N takes $m\lceil n/m \rceil + \lceil k/m \rceil$. For n large enough,

$$m\lceil n/m \rceil \leqslant (1+\varepsilon)n$$

which implies that eventually

$$f_N \leqslant \lceil f_M/m \rceil + m\lceil n/m \rceil$$

If we choose m large enough, we obtain the inequality desired. □

If we only use one tape, then the increase becomes $\varepsilon f_M + 2n^2 + 2$. We also note that programs which run in linear time can only improved to $(1+\varepsilon)n$, for some ε. This makes sense, for if some program runs in time bounded by cn for $c < 1$, then for large inputs the program must eventually not even look at all the input, which is unfeasible in most problems.

Since we do not describe Turing machines formally, we rely on heuristic requirements to determine upper bounds for certain algorithms. Our intuition, guided by our knowledge of formal Turing machines, should carry us through, provided we describe algorithms in enough detail that the underlying Turing machine can be seen in enough detail.

We now have the formality to classify problems by how long it takes to compute them. Given a function g, define $\mathbf{TIME}(g)$ to be the class of all languages L which are recognized by a machine M, and $f_M \in O(g)$. Thus $\mathbf{TIME}(g)$ consists of all problems which can be computed 'roughly in time with g'. This is the first example of a **complexity class**, a set of languages characterized by how slow it takes to decide upon the language.

Example. *Consider the language $A = \{0^k 1^k : k \geqslant 0\}$. Determine a turing machine deciding the language from the algorithm*

1. *Scan across input, checking whether the input is of the form $0^i 1^j$ for some i, j. We ensure the string only contains 0s and 1s, and that the string contains no 0s after it contains 1s. Reject in any other circumstance. Afterwords, return to the beginning of the tape.*

2. *While there are still unmarked 0s and 1s on the tape, tick off a new 0 on the tape and a new 1. If there are no 0s but still 1s, or 0s but no 1s, reject the input.*

3. *We have ticked off one 0 for each 1, so there are the same number of zeroes as ones. Accept the input.*

We analyze the three steps individually, putting the asymptotics together once we're done. Step 1 takes roughly a constant number of steps for each element of the input, for we perform the same operation on each character. Thus the time to compute step 1 is in $O(n)$. The number of times step 2 is executed is at most half the characters in the input, and each iteration is in $O(n)$, for the iteration involes moving from the beginning to the end of the tape a finite number of times. Thus step 2 is in

$$O(n/2)O(n) = O(n)^2 = O(n^2)$$

Finally, step 3 is a constant time operation, and is therefore in $O(1)$. Thus the entire algorithm is in

$$O(n) + O(n^2) + O(1) = O(n^2 + 2n + 1) = O(n^2)$$

Thus $A \in \mathbf{TIME}(n^2)$. A divide and conquer variation of this algorithm shows that $A \in \mathbf{TIME}(n \log n)$, left as an exercise.

8.2 Models of Complexity

We have considered various variants of Turing machines. All turn out to decide the same class of languages, hence the adoption of the machine as an applicable model of real world computation. But we run into issues when applying this argument to complexity theory, for using a different model of turing machine may reduce the time complexity of an algorithm.

Example. *Consider the $0^k 1^k$ problem. The best algorithm we found ran in $O(n \log n)$ time. But consider a 2-tape Turing machine, which first shifts all 1s in the input to a 2nd tape, then procedurally checks off 0s and 1s. This runs in $O(n)$ time, for we need only scan over the tape a constant number of times. Later, we shall show that the asymptotics of the $0^k 1^k$ problem cannot be improved on a single tape machine, so multi-tape turing machines are asymptotically faster than one tape machines.*

We may take comfort in discovering that changing the model does not *drastically* affect the computation time of an algorithm. This is what this section sets out to address.

Theorem 8.2. *If a multi-tape turing machine has time complexity f, where $f \geqslant n$, then the multi-tape turing machine has an equivalent single-tape turing machine with time complexity in $O(f^2)$.*

Proof. We have already described a procedure which simulates a k-tape turing machine. We shall compute an asymptotic analysis of this simulation. Suppose the multitape turing machine takes $g(n)$ steps before terminating on a particular input of size n. Let us analyze each step

1. The algorithm first takes the input $w_1 \dots w_n$, and manipulates it into the form
$$\# \dot{w}_1 \dots w_n \# \dot{\sqcup} \# \dot{\sqcup} \# \dot{\sqcup} \# \dots \# \dot{\sqcup}$$
where we have k hashtags. This takes $O(n+k)$ steps.

2. For each of the $g(n)$ steps the multitape machine takes, we must pass through our simulation tape, recognizing the current state. We then perform a second pass to move dots left or right when needed. In each step, we add at most k new elements to our tape, so the size of the tape is always bounded by $n + kg(n)$. Therefore the number of steps in the first pass is in $O(n + kg(n))$. The second pass moves through the tape, and pushes at most k new symbols into the tape, otherwise just moving the dots in the tape back and forth. A push moves at most $n + kg(n)$ symbols to the right, and thus the number of steps we take is in $O(k(n + kg(n)))$.

Step 2 is applied $g(n)$ times, so overall, the speed of the algorithm is in

$$O(n+k) + O(n+kg(n)) + g(n)O(n+kg(n)) = O(ng(n) + g^2(n))$$

assuming that $n \in O(g)$, then the speed is in $O(g^2)$. □

Thus, though multitape machines may compute faster, they only introduce do things quadratically faster than single tape machines. One of the biggest issues with complexity theory is that this is not true of non-deterministic machines.

First off, we must debate how long a non-deterministic machine takes to compute. We cannot simply count all steps the non-deterministic machine takes, because the machine takes many branches, some of which may be infinite. Since we can see non-deterministic computation as some sort of parallel processing, we could define the time to be the length of the shortest branch of processesing which yields termination. Mathematically, however, we will see that it is more convenient to define the run time to be the length of the longest branch. We are then able to define the time complexity of a non-deterministic automata.

Theorem 8.3. *If a non-deterministic machine runs in $O(f)$ time, then there is an equivalent deterministic automata which runs in $2^{O(f)}$ time.*

Proof. Perform a time analysis on the turing machine which simulates a non-deterministic machine. □

Thus non-deterministic algorithms are fundamentally connected to exponential deterministic algorithms. This makes sense, for in general, if a problem can be divided into exponentially many cases to check, each verifiable in a linear amount of time, then a non-deterministic algorithm can split into each possible case, exponentially dividing, and then check each case in a polynomial amount of time, giving us a polynomial time algorithm. Thus the time of Non-deterministic turing machines is bounded by the time it takes to verify a single case.

Now we get to the fun stuff. Define the complexity class

$$\mathbf{P} = \bigcup_{k=0}^{\infty} \mathbf{TIME}(n^k)$$

these are all problems computable in polynomial time on a deterministic automata.

Chapter 9

The PCP Theorem, and Probabilistic Proofs

The Cook-Levin discovery of NP completeness hints that there are a certain class of **NP** problems which are computationally much more complex than other problems. In particular, the theorem says that if the Boolean satisfiability problem SAT has a polynomial time algorithm, then all problems in **NP** are verifiable in polynomial time, solving the $\mathbf{P} = \mathbf{NP}$ conjecture in the affirmative. Since most computer scientists do not believe that $\mathbf{P} = \mathbf{NP}$, we expect SAT not to have a polynomial time solution. In the face of this computational wall, it is natural to explore other strategies to 'solving' the satisfication problem.

Rather than considering an algorithm which is completely correct, we instead try to find algorithms which are 'approximately correct'. Consider the MAX 3-SAT problem, in which given a 3-SAT instance x, we are asked to find a truth assignment which maximizes the fraction of clauses that can be satisfied. We call this the value of x, denoted $\text{val}(x)$. We say an algorithm is a ρ-approximation for MAX-3SAT if the output of the algorithm given an input x consisting of m clauses is a truth assignment satisfying at least $\rho \cdot m\text{val}(x)$ of the clauses. It is an interesting question to ask whether there is a boundary in how well MAX-3SAT can be approximated. That is, are there polynomial time algorithms which can approximate MAX-3SAT to an arbitrary degree of precision?

The PCP theorem essentially answers this in the negative. In short, the theorem shows that there is $\rho < 1$ such that for every language $L \in \mathbf{NP}$, there is a polynomial-time computable function f mapping strings in the alphabet of L to $3SAT$ instances such that if $x \in L$, then $\text{val}(f(x)) = 1$, and if $x \notin L$, then $\text{val}(f(x)) < \rho$. Thus the PCP theorem immediately implies that if there are ρ-approximation algorithms for MAX-3SAT for ρ arbitrarily close to 1, then $\mathbf{P} = \mathbf{NP}$. Thus the theorem essentially guarantees the existence of problems which are 'hard to approximate'.

Is is natural to try and adapt the standard proof of the Cook-Levin theorem to proving that this claim is true. Since the 1970s, when the theorem was discovered, researchers focuses on this reduction, but soon discovered that these methods do not suffice. The good news is that the theorem does construct a polynomial-time computable function f mapping strings over an alphabet representing an NP problem L into a Boolean function such that $\text{val}(f(x)) = 1$ if and only if $x \in L$, but the construction does not create a 'gap' in the values of $f(x)$ for $x \notin L$ – indeed for most problems we find that there are values $\text{val}(f(x))$ which are arbitrarily close to 1, with $x \notin L$. The PCP theorem therefore finds a truly novel encoding of problems as Boolean satisfaction problems, which are easy enough to approximate that approximating MAX-3SAT is difficult.

9.1 PCP and Proofs

There is another way to look at the PCP theorem from the perspective of formal computability theory. Recall that the class **NP** consists of problems whose solutions can be verified in polynomial time. That is, a language L is in **NP** if there is a Turing machine taking an input x and certificate π that runs in time polynomial to the input x, and such that $x \in L$ if and only if the turing machine accepts an input (x, π) for some certificate π. If we see π as a 'proof' of the validity of x, then an NP problem is one whose proofs can be effectively checked.

Suppose that we weaken this condition, considering randomized algorithms which reject invalid proofs with high probability. We would expect that these algorithms form a class much more general than **NP** – the PCP theorem says that if we restrict access to the verification certificate, and to the amount of randomness that a function can have, then this new classification of problems is no more general than **NP**. This also implies that certificates to problems in **NP** can be encoded in a way that solutions require little access to the certificate. Thus the theorem gives bounds on the amount of randomness and the amount of checking required to distinguish certain encodings of **NP** problems.

If we are to restrict a machines access to the certificate π, we must distinguish between the types of access a machine could have. **Non-adaptive** queries are made based only on the input x, and perhaps some randomness, whereas **adaptive queries** are allowed to depend on previous inputs. We will restrict ourselves to non-adaptive queries, because these have effective derandomization properties, but the question of allowing adaptive queries is still interesting.

So let's define the notion of a randomized certificate checker. Given a language L, and functions $f, g : \mathbf{N} \to \mathbf{N}$ we define a (f, g) **PCP verifier** of L to be a probabilistic Turing machine, running in polynomial time in x which, given an input x of length n, and given random access to a certificate π (called the 'proof' of x), accepts or rejects x based on at most $g(n)$ random bits and $f(n)$ non-adaptive queries to locations of π. We require that if $x \in L$, then there is a proof π such that the machine is certain to accept x, in which case we call π the **correct proof**, or if $x \notin L$, then for *every* proof π, the machine rejects x with a positive probability which is bounded below by a constant independant of the input. We define the language $\mathbf{PCP}(f, g)$ to be the space of problems which have a (f, g) verifier. More generally, for classes C, C' of functions (like $O(n)$, $\text{poly}(n)$, etc), we define the language $\mathbf{PCP}(C, C')$ to be the class of problem with an (f, g) verifier, where $f \in C$ and $g \in C'$.

We note that the probability with which a false x is rejected can be set to any constant value, and is normally set such that every false input is rejected with probability greater than or equal to $1/2$. The reason for this generality is that if a problem L has a (f, g) verifier rejecting every $x \notin L$ with probability $1 - P$, and by running this verifier independently n times, we obtain a (nf, ng) verifier which rejects x with probability $1 - P^n$, and has the same asymptotic querying properties. Furthermore, we might as well assume that the length of any certificate π we consider has length $f(n)2^{g(n)}$, because an algorithm using k random bits to choose l non-adaptive queries can only choose between $l2^k$ possible locations to query from.

If we have a PCP verifier for a problem L, using $f(n)$ points of access to the certificate, and $g(n)$ random bits, then we may design a Turning machine which, given x and π, simulates the PCP verifier on all $2^{g(n)}$ possible random inputs, calculates the probability of accepting x, and rejects x if the probability than x is rejected is greater than the specified constant.

This Turing machine verifies L precisely, and runs in time $O(2^{g(n)+O(\log n)})$.

We can guess π by nondeterministically choosing $f(n)2^{g(n)}$ symbols for π, and then running the computer on all these inputs. Thus if L is an (f,g) verifiable problem, then L is also a problem in $\mathbf{NTIME}(f(n)2^{g(n)+O(\log n)})$, the space of problems solvable by a nondeterministic Turing machine in a specified amount of time. In particular, this implies that any problem in $\mathbf{PCP}(O(\log n), O(1))$ can be solved by a nondeterministic Turing machine running in time $O(\log n)2^{O(\log n)}$, hence $\mathbf{PCP}(\log n, 1) \subset \mathbf{NP}$. The PCP theorem is the converse to this statement.

Theorem 9.1 (The PCP Theorem). $\mathbf{NP} = \mathbf{PCP}(O(\log n), O(1))$

We therefore obtain the surprising fact that there is a particular encoding of the certificates of **NP** problems which enable the problem to be verified with high probability restricting an algorithm to logorithmic randomness, and only checking a constant number of symbols in some proof of a solution. This leads to surprising results. For instance, given an axiomatic system in which proofs can be verified in polynomial time to the size of the proof, the language

$$\{\langle x, 1^n \rangle : x \text{ is a formula with a proof of length at most } n \text{ characters}\}$$

is in **NP**. It follows that we can encode the proofs in this axiom system in such a way that we can check the correctness of an encoded proof in polynomial time with randomness linear in the size of the proof, and only looking at a constant number of bits!

Example. *Consider the graph non-isomorphism problem, which asks us to determine, given a pair of graphs $\langle G_0, G_1 \rangle$, each containing n nodes, whether G_0 is not isomorphic to G_1. We claim the problem is in $\mathbf{PCP}(poly(n), O(1))$. First, we index all graphs with n nodes by an integer between 1 and 2^{n^2}, so that certificates π of length 2^{n^2} can be considered as Boolean valued maps from the set of all graphs with n nodes. A 'true' proof π that G_0 is not isomorphic to G_1 shall let $\pi(X) = 0$ if X is isomorphic to G_0, $\pi(X) = 1$ if X is isomorphic to G_1, and $\pi(X)$ chosen arbitrarily if neither holds. Our verifier picks an index $i \in \{0,1\}$, and a permutation $\nu \in S_n$, uniformly at random, using $1 + \log(n!) = O(n \log n)$ random bits. We use ν to rearrange the vertices of G_i to obtain an isomorphic graph $\nu(G_i)$. We accept if $\pi(\nu(G_i)) = i$. If we have a correct proof π, and G_0 is not isomorphic to G_1, then the verifier always accepts, as required. If G_0 is* *isomorphic to G_1, then the probability distribution of $\nu(G_i)$ is independent of i, and therefore $\mathbf{P}(\pi(\nu(G_i)) = i) = 1/2$, hence the verifier rejects the instance with probability $1/2$.*

It turns out that $\mathbf{PCP}(\text{poly}(n), O(1)) = \mathbf{NEXP}$, so that the verifier construction above is a special case of a more general theorem, but we will not prove be able to build the theory to the sophistication which can prove this equality, so we leave it for another time.

9.2 Equivalence of Proofs and Approximation

The two interpretations of the PCP theorem are, of course, equivalent. This becomes clear when we introduce the family of k constraint satisfaction problems, denoted k-CSP. In the problem, we take a set of k-juntas $f_1, \dots, f_m : \{0,1\}^n \to \{0,1\}$, called the constraints. An assignment $x \in \{0,1\}^n$ satisfies the constraints if $f_i(x) = 1$ for each i. In general, we define the value of the constraints f_i to be the maximum fraction of the constraints that can be satisfied, denoted $\text{val}(f)$. The 3-SAT problem is a subproblem of the 3-CSP problem, where each f_i is a disjunction of variables.

Now given a natural number k and $\rho \leqslant 1$, define the YES ρ-GAP k-CSP problem to be the problem of determining, given an instance f of k-CSP, whether $\text{val}(f) = 1$, and the NO ρ-GAP k-CSP problem to determine whether $\text{val}(f) < \rho$. We say ρ-GAP k-CSP is **NP**-hard if, for any **NP** problem L, there is a polynomial time function g mapping strings in L to strings for the ρ-GAP k-CSP problem, such that if $x \in L$, then $\text{val}(g(x)) = 1$, and if $x \notin L$, then $\text{val}(g(x)) \leqslant \rho$. There is a natural equivalent statement of the PCP theorem in the context of k-CSP problems.

Theorem 9.2. *There is k and $\rho \in (0,1)$ such that ρ-GAP k-CSP is* **NP** *hard.*

To see the equivalence, suppose that $\mathbf{NP} = \mathbf{PCP}(O(\log n), O(1))$. We will show $1/2$-GAP k-CSP is **NP** hard for some k. It is sufficient to reduce 3-SAT to a problem of this type, because we can reduce all problems in **NP** to 3-SAT in polynomial time. Because of the assumption, there is a PCP verifier for 3-SAT using $K \log n$ random bits and K' queries to the certificate. Given any input x to 3-SAT and $r \in \{0,1\}^{K \log n}$, let f_{xr} be the function that takes as input a proof π, and outputs 1 if the verifier accepts π on input x and random bits r. Note that f_{xr} only depends on at most K' entries of π. For every $x \in \{0,1\}^n$, the collection of all f_{xr} is describable with

$K'n^K$ bits, and since each f_{xr} runs in polynomial time, we can transform an input x to the set of all f_{xr} in polynomial time in the size of x. If x is satisfiable, then $\text{val}(x) = 1$, whereas if x is not satisfiable, then $\text{val}(x) \leqslant 1/2$. Thus we have reduced 3-SAT to $1/2$-GAP K'-CSP in polynomial time, and therefore the $1/2$-GAP K'-CSP problem is **NP** hard.

Conversely, suppose ρ-GAP k-CSP is **NP** hard for some ρ and k. Given any **NP** problem L, consider a reduction g to ρ-GAP k-CSP with constraints f. Consider the following PCP verifier for an instance of k-CSP. We shall interpret a certificate π as an assignment to the variables x_i such that all constraints are satisfied. Our algorithm will take a random index i, and check whether $f_i(x) = 1$ (we need only access k bits of π to determine this). If $x \in L$, then $g(x)$ will always be accepted when π is the correct proof. If $x \notin L$, then $\text{val}(g(x)) < \rho$, hence the probability of $g(x)$ being accepted by the verifier is less than ρ. This implies that L is in $\mathbf{PCP}(O(\log n), O(1))$.

Now suppose that 3-SAT is hard to approximate, for $\rho > 0$, where for every $L \in \mathbf{NP}$, there is a reduction f of L to 3-SAT such that $\text{val}(f(x)) = 1$ if $x \in L$, and $\text{val}(f(x)) < \rho$ if $x \notin L$. This is essentially a special case that ρ-GAP 3-CSP is **NP** hard. To complete the equivalence, it suffices to verify that the **NP** hardness of ρ-GAP k-CSP reduces to the **NP** hardness of the more general ρ'-GAP 3-CSP problem, for perhaps a slightly relaxed $\rho' > \rho$. If $\rho = 1 - \varepsilon$, let $f_1, \dots, f_n$ be a k-CSP instance with n variables and m constraints. Because each f_i is a k-junta, it can be expressed as the conjunction of at most 2^k clauses, where each clause is the or of at most k variables or their negations. Let X be the set of all such constraints, bounded in size by $m2^k$. If f is a YES instance of ρ-GAP k-CSP, then there is a truth assignment satisfying all of the clauses of X. If f is a NO instance, then every assignment violates at least ε of the conditions f_i, and therefore $\varepsilon/2^k$ of the clauses of X. Using the Cook-Levin technique, we can transform any clause C on k variables to k clauses $C_1, \dots, C_k$ over the variables $x_1, \dots, x_k$ with additional variables $y_1, \dots, y_k$ such that each C_i has only 3 variables, and if the x_i satisfy C, then we may pick y_i to satisfy all C_i simultaneously, and if the x_i cannot satisfy C, then for every choice of y_i some C_j is not satisfied. Let Y denote the 3-SAT instance of $km2^k$ clauses over the new set of $n + km2^k$ variables obtained from X. If X is satisfiable, so is Y, hence $\text{val}(Y) = 1$, and if X is not satisfiable, then every assignment verifies at least $\varepsilon/k2^k$ of the constractions of Y. Thus $\text{val}(Y) \leqslant 1 - \varepsilon/k2^k$, and thus we have reduced L to MAX 3-SAT with a $1 - \varepsilon/k2^k$ separation.

9.3 NP hardness of approximation

As the Cook-Levin theorem gives us a whole family of NP-complete problems, the PCP theorem gives us hardness of approximation results for many more problems than 3SAT and CSP. As an example, we consider the approximation problem for the MAX-INDSET problem, which asks us to find the maximum independent set in a grpah (the largest number of vertices sharing no edge), and the MIN-COVER problem, which asks us to find the minimum set of vertices on a graph such that every vertex in the graph is connected by an edge to this set.

The complement of every independent set is a cover, and the complement of every cover is an independent set, so that MIN-COVER is equivalent to MAX-INDSET as exact problem classes. This does not extend to a more robust equivalence. If C is the size of the minimum vertex cover, and I the size of the maximum independent set, then $C + I = n$. Therefore, if we have a ρ approximation algorithm for MAX-INDSET, returning a set S with $\rho I \leqslant S$, then we would find a corresponding cover of size

$$n - S \leqslant \frac{n - \rho I}{n - I}(n - I) = \frac{n - \rho I}{C} C$$

and we therefore we have an approximation algorithm for the MIN-COVER problem of ratio $(n - \rho I)/C$. A similar relationship exists between transforming approximation algorithms of MIN-COVER to MAX-INDSET. However, it turns out that the approximation ratios of these two problems are very different – there is no constant-value approximation algorithm for the MAX-INDSET problem, whereas there is a trivial $1/2$ approximation for MIN-VERTEX-COVER.

Lemma 9.3. *There is a polynomial time reduction f from 3-SAT formulas containing n clauses to graphs such that $f(X)$ is an $7n$-vertex graph whose largest independant set is $n \cdot val(X)$.*

Proof. Consider the standard exact reduction of 3-SAT to independent set, which takes a logical equation X with m clauses, and returns a graph $f(X)$ with $7m$ vertices, such that there is an assignment satisfying k clauses of X

if and only if $f(X)$ has an independent set of size k. We do this by associating with each clause C seven nodes of the form C_{xyz}, where $(x, y, z) \in \{0, 1\}^3$ is a truth assignment of the variables in the clause which satisfy the clause (there are 7 total possible assignments). Given two clauses C^1 and C^2, put an edge between $C^1_{x_0 y_0 z_0}$ and $C^2_{x_1 y_1 z_1}$ if the assignments are incompatible. Certainly all 7 nodes connected to a single clause are connected. An independent set in this graph of size k is then a set of consistent assignments satisfying at least k clauses, and conversely, a consistent satisfaction of k clauses leads to an independent set in the graph of size k. $\square$

Theorem 9.4. *There is $\gamma < 1$ and $\lambda > 1$ such that if there is a γ-approximation algorithm for MAX-INDSET, or a λ approximation algorihtm for MIN-COVER, then* $\mathbf{P} = \mathbf{NP}$.

Proof. Let L be an **NP** language. The PCP theorem implies there is a polynomial time computable reduction f to MAX-3SAT. That is, for some $\rho < 1$, X is satisfiable and $\text{val}(f(X)) = 1$, or X is not satisfiable and $\text{val}(f(X)) < \rho$. The last theorem implies that if we had a ρ-approximation to IND-SET, then we could do a ρ-approximation on MAX-3SAT, thereby proving $\mathbf{P} = \mathbf{NP}$.

For MIN-VERTEX-COVER, the minimum vertex cover of the graph obtained in the previous lemma has size $n[1 - \text{val}(X)/7]$. Hence if MIN-VERTEX-COVER had a ρ' approximation algorithm for $\rho' = (7 - \rho)/6$, then we could find a vertex cover of size $\leqslant \rho' n(1 - 1/7) \leqslant n(1 - \rho/7)$ if $\text{val}(X) = 1$, hence we could find a MAX-IND-SET of size greater than $n\rho/7$, and this shows for ρ close enough to 1, a ρ' approximation to MIN-VERTEX-COVER would imply $\mathbf{NP} = \mathbf{P}$. $\square$

It turns out that MAX-INDEPENDENT-SET has an ‘amplification procedure’, which can generate constant time approximation algorithms of arbitary accuracy, given the existence of a particular solution. This is given by the ‘graph product’ technique. Given a graph G with n nodes,let G^k be the graph on $\binom{n}{k}$ vertices corresponding to subset of nodes in G of size k. Define two subsets S and T to be adjacent if $S \cup T$ is an independent set. The largest independent set in G^k consists of all k-size subsets of the largest independent set in G, and therefore has size $\binom{I}{k}$, where I is the maximum independent set in G^k. Thus if a ρ approximation algorithm for the MAX-INDEPENDENT-SET problem implies $\mathbf{P} = \mathbf{NP}$, then a

$$\frac{\binom{\rho I}{k}}{\binom{I}{k}} = \frac{(\rho I)(\rho I - 1)\dots(\rho I - k + 1)}{I(I-1)\dots(I-k+1)} = \rho^k \prod_{m=0}^{k-1} \frac{I - m/\rho}{I - m} \leqslant \rho^k$$

approximation algorithm for k powers of graphs implies $\mathbf{P} = \mathbf{NP}$. For k large enough, we find that the existence of any constant time algorithm for MAX-INDEPENDENT-SET implies $\mathbf{P} = \mathbf{NP}$.

9.4 The Proof of PCP

We shall now prove a weak form of the PCP theorem, which can be used to attack the finer problem. First, we consider the notion of Walsh-Hadamard encoding. Given a string $x \in \mathbf{F}_2^n$, we define the **Walsh-Hadamard** encoding to be the string $W(x) : 2^{[n]} \to \mathbf{F}_2$ such that $W(x)(S) = \sum_{i \in S} x_i$. The encoding is essentially the tuple representation of $x^* : \mathbf{F}_2^n \to \mathbf{F}_2$, where $x^*(y) = \sum x_i y_i$. The encoding is very inefficient, since elements of $\mathbf{F}_2^n$ are length two strings, but we shall find the encoding is very useful to represent proof certificates of certain decision problems over the field $\mathbf{F}_2$, because it enables us to compute the inner product of x and y with access to a single bit of the proof certificate.

Theorem 9.5. $\mathbf{NP} \subset \mathbf{PCP}(\mathit{poly}(n), O(1))$.

Proof. We will consider a particular $\mathbf{NP}$ complete problem, and show it has a verifier which is $(\text{poly}(n), O(1))$. Since every $\mathbf{NP}$ is exactly reducible to this $\mathbf{NP}$ complete problem, this suffices to prove the full inclusion. The $\mathbf{NP}$ complete problem we will use is the problem QUAD-SAT, which asks to solve solve a system of m quadratic equations over $\mathbf{F}_2^n$. This problem is NP complete – it is most easiest to see this by reducing the problem of $CKT-SAT$, determing whether a Boolean circuit is satisfiable, expressing AND and OR operations by quadratic polynomials. We may assume that each term of the system contains a pair of terms, since $x^2 = x$ in $\mathbf{F}_2$. If the equations are

$$\sum a_{ij}^k x_i x_j = b_k$$

where k ranges from 1 to m, then we see that we can encode the problem as a '$m \times n^2$' matrix A, and $b \in \mathbf{F}_2^m$. If we consider the n^2 dimensional vector $(x \otimes x)_{ij} = x_i x_j$, for $x \in \mathbf{F}_2^n$, then we can write the equation as $A(x \otimes x) = b$.

Now suppose that an instance (A, b) of QUAD-SAT is satisfiable by some vector x. We will interpret a proof π as a pair of functions

$$f : \mathbf{F}_2^n \to \mathbf{F}_2 \qquad g : \mathbf{F}_2^{n \times n} \to \mathbf{F}_2$$

where f is the Walsh-Hadamard encoding of x, and g is the Walsh Hadamard encoding of $x \otimes x$. The encoding enables us to calculate values $\sum_{i \in S} x_i$ and $\sum_{(i,j) \in S} x_i x_j$ with only a single query to the certificate. The proof consists of a certain number of steps

1. First, we use the certificate to verify that f and g are linear. Otherwise, the proof is not correctly encoded. We random choose whether to test f and g, and then using three queries, we test either of the functions. If f is ε far away from being linear, or g is ε far away from being linear, the test is rejected with probability $(1 - \varepsilon)/2$. Otherwise, we may assume from now on that f and g are ε-close to being linear, and we may use local decoding to calculate values of the linear function f and g approximate with high probability. Thus let $d(f, \tilde{f}) < \varepsilon$, and $d(g, \tilde{g}) < \varepsilon$. Since $\tilde{f}$ and $\tilde{g}$ are linear, they actually encode elements of $\{0,1\}^n$ and $\{0,1\}^{n^2}$ respectively.

2. Check if $\tilde{g}$ encodes $x \otimes x$, where $\tilde{f}$ encodes x. To do this, if the equality held, then we would find

$$\tilde{f}(y)\tilde{f}(z) = \langle x, y \rangle \langle x, z \rangle = \sum_{ij} x_i x_j y_i z_j = \langle x \otimes x, y \otimes z \rangle = \tilde{g}(y \otimes z)$$

   If we choose $Y, Z \in \{0,1\}^n$ uniformly at random, and check whether $\tilde{f}(Y)\tilde{f}(Z) = \tilde{g}(Y \otimes Z)$ using 9 queries of the certificate. If $\tilde{g}$ does not encode $x \otimes x$, then $x \otimes x - \tilde{g}$ is a non-zero linear functional, hence it takes value 0 on half the inputs, and value 1 on the other half, so

$$\mathbf{P}(\tilde{f}(Y)\tilde{f}(Z) = \tilde{g}(Y \otimes Z)) = 1/2$$

   Assuming f is ε close to $\tilde{f}$, and g is ε close to $\tilde{g}$, we calculate $\tilde{f}$ and $\tilde{g}$ correctly with probability $1 - 2\varepsilon$ each. Thus we find that if $\tilde{g}$ does not encode $x \otimes x$, then we reject with probability at least $(1 - 2\varepsilon)^2/2$.

3. All that remains is to check that $\tilde{g}(A^k) = b_k$ for each k. To do this for each k would take far too many queries to the certificate. Thus we pick a random $z \in \mathbf{F}_2^n$, and then determine if

$$\langle z, \tilde{g}(A) \rangle = \sum z_i \tilde{g}(A_i) = \sum z_i b_i = \langle z, b \rangle$$

If $\tilde{g}(A) \neq b$, then provided we pick Z uniformly $\langle Z, \tilde{g}(A) \rangle \neq \langle Z, b \rangle$ with probability $1/2$, and therefore we reject with probability at least $(1-2\varepsilon)/2$, with only four queries to the certificate.

In conclusion, with 16 queries of the certificate, we reject an incorrect proof with probability greater than or equal to

$$\min\left(1-2\varepsilon, \frac{1-2\varepsilon}{2}, \frac{(1-2\varepsilon)^2}{2}\right) = \frac{(1-2\varepsilon)^2}{2}$$

And, with a more stringent rejection probability (randomly performing one of the steps above, not all of them in sequence), we can perform the test with only 9 queries. □

9.5 Håstad's theorem and Explicit Hardness of Approximation Bounds

If the PCP theorem holds, we can construct a $(O(\log n), O(1))$ verifier for any problem in **NP**. However, it is still of interest to try and find explicit bounds on randomness and bit query numbers, rather than asymptotic bounds. This is not just for academic interest, because bounds on the number of bits used gives explicit hardness of approximation bounds on certain problems. Trying to understand these explicit bounds corresponds to the more advanced PCP theorems which have been proven. It was Håstad who found that we can obtain essentially the same result as the PCP theorem with only three queries.

Theorem 9.6. *For every $\delta > 0$ and every* **NP** *language L, there is a $(\log(n), 3)$ PCP verifier for L, such that for every $x \in L$, there is a proof certificate π such that the PCP verifier accepts with probability greater than or equal to $1-\delta$, and for every $x \notin L$ and every incorrect proof π, the PCP verifier rejects x with probability greater than or equal to $1/2 - \delta$. What's more, the PCP verifier just takes a proof π of length m, chooses $i, j, k \in [m]$ and $b \in \{0,1\}$ randomly according to some distribution, and accepts if $\pi_i + \pi_j + \pi_k = b$ (modulo 2).*

It is not superfluous that the PCP verifier uses a linear equation as a PCP verifier. Without further work, the standard PCP theorem only allows us to show that problems are hard to approximate up to some unspecified constant. But Håstad's theorem enables us to get a precise approximation bound for the problem MAX-E3LIN, which takes as input a sequence of linear equations $x_i + y_i + z_i = b_i$, where x_i, y_i, z_i are variables, and $b_i \in \mathbf{F}_2$, and tries to find the maximal subset of equations which can be simultaneously satisfied over $\mathbf{F}_2$. Note that unlike the E3LIN, which asks to decide whether all linear equations can all be simultaneously solved, MAX-E3LIN is NP complete. Håstad shows that there is a known constant such that, if we can approximate MAX-E3LIN past this ratio, the $\mathbf{P} = \mathbf{NP}$.

Theorem 9.7. *If there is a* $1/2 + \rho$ *approximation algorithm for MAX-E3LIN, for any* $\rho > 0$, *then* $\mathbf{P} = \mathbf{NP}$.

Proof. The PCP verifier that Håstad constructed implies that any problem L is equivalent to an instance of MAX-E3LIN with $2^{O(\log n)} = \text{poly}(n)$ equations, where either $1 - \delta$ of the equations are solvable, or at most $1/2 - \delta$ of the equations are. If we had a μ approximation algorithm for MAX-E3LIN, where if we can satisfy C of the clauses, then our algorithm returns an assignment satisfying greater than or equal to μC of the clauses. Provided that μC is greater than $1/2 - \delta$ whenever $C \geqslant 1 - \delta$, then we can apply the PCP verifier reduction to solve any **NP** problem in polynomial time. Thus if $\mathbf{P} \neq \mathbf{NP}$ we find $\mu(1-\delta) < 1/2 - \delta$, so that $\mu < (1/2 - \delta)/(1-\delta)$, and since δ is arbitrary, we can let $\delta \to 0$ to conclude that $\mu \leqslant 1/2$. ☐

This is a tight threshold result, as we know of algorithms which approximate MAX-E3LIN with a $1/2$ ratio.

Theorem 9.8. *There is a 1/2 approximation algorithm for MAX-E3LIN.*

Proof. TODO ☐

Using the PCP theorem, we can reduce MAX-E3LIN to MAX-3SAT robustly, in the sense that approximation algorithms to MAX-E3LIN get converted to approximation algorithms to MAX-3SAT. Thus we obtain hard approximation bounds for MAX-3SAT.

Theorem 9.9. *If there is a* $7/8 + \rho$ *approximation algorithm for MAX-3SAT, for any* $\rho > 0$, *then* $\mathbf{P} = \mathbf{NP}$.

Proof. Consider determining if more than $1 - \delta$ of the clauses in MAX-E3LIN are satisfied, or if at most $1/2 + \delta$ of the clauses can be satisfied. For $a, b, c \in \mathbf{F}_2$, $a + b + c = 0$ holds if and only if

$$\neg a \vee b \vee c \qquad a \vee \neg b \vee c \qquad a \vee b \vee \neg c \qquad \neg a \vee \neg b \vee \neg c$$

are simultaneously satisfied. Similarly, $a + b + c = 1$ holds if and only if

$$a \vee b \vee c \qquad \neg a \vee \neg b \vee c \qquad \neg a \vee b \vee \neg c \qquad a \vee \neg b \vee \neg c$$

hold simultaneously. Thus if the original E3LIN instance contains n equations, then the constructed instance of 3SAT contains $4n$ clauses, and if m of the E3LIN equations can be satisfied, then at most $4m+3(n-m) = m+3n$ of the clauses of the converted 3SAT problem can be satisfied. Thus deciding whether $1-\delta$ of the equations in MAX-E3LIN can be satisfied, or whether less than $1/2+\delta$ of the equations can be satisfied reduces to determining whether more than $(1/2+\delta)n+3n$ of the equations of the 3SAT problem can be satisfied, so unless $\mathbf{P} = \mathbf{NP}$, if we have a μ approximation algorithm for 3SAT, then $4\mu n < (1/2+\delta)n+3n$, hence $\mu < 7/8+\delta/4$, and we can let $\delta \to 0$ to conclude $\mu \leqslant 7/8$. □

Note that this theorem only requires the use of 3SAT where each clauses has exactly three variables, so this version of the problem is equally hard. What's more, there are elementary algorithms for 3SAT with exactly three variables which give 7/8 approximation algorithms, so that this is a tight bound on the approximation.

Theorem 9.10. *There is a* $7/8$ *approximation algorithm for 3SAT.*

Proof. TODO □

To prove Håstad's theorem, we need to discuss the constraint satisfaction problem on a more general alphabet than the boolean satisfaction we discussed over the PCP theorem. Given a finite alphabet W of ize n and a collection of m-juntas functions $f_i : W^n \to \{0,1\}$, we ask if we can find an assignment $x \in W^n$ which maximizes the number of f_i with $f_i(x) = 1$. We let $\text{val}(f)$ denote the maximum fraction of the constraints f which can be satisfied. This is the m-CSP_n problem. As with the proof of the equivalence of the PCP theorem to certain hardness of approximation results for 3SAT, we introduce the YES ρ-GAP m-CSP_n problem as determining whether a given constraint problem f has $\text{val}(f) = 1$, and the NO ρ-GAP m-CSP_n problem as determining whether a given constraint problem has $\text{val}(f) < \rho$, over a fixed language W of size n. The next theorem is a key component of the PCP theorem.

Theorem 9.11. *ρ-GAP 2-CSP_W is hard to approximate for some $0 < \rho < 1$ and some alphabet W.*

We obtain Håstad's result by showing that we can decrease our choice of ρ without increasing the size of W too much, while still maintaining the approximation hardness of the gap. The importance of the CSP in the theorem of Håstad is because of the following reduction, due to Ran Rad.

Theorem 9.12. *There is a $c > 1$ such that for every $t > 1$, 2^{-t}-GAP 2-CSP$_{2^{ct}}$ is* **NP** *hard, and we need only work over instances of 2-CSP$_n$ which are* **regular**, *and have the* **projection property**, *in the sense that every variable is relevant in the same number of constraints, and for every constraint f_i, and each $x \in W$, there is a unique y for which $f_i(x,y) = 1$.*

Thus it suffices to find a $(O(\log n), 3)$ verifier for 2-CSP$_n$ which is sufficiently accurate to maintain the gap constructed in the theorem, so that we can extend this verifier to all **NP** problems.

If every **NP** problem can be reduced to a $(O(\log n), 3)$ verifier, then $(1/2)$-GAP 3-CSP instance is **NP** hard, because we can reduce 3SAT to a version of this problem with poly(n) constraints. Conversely, the **NP** hardness of $(1/2)$-GAP 3-CSP

We also rely on a fact that is a key part of the path to proving the full PCP theorem.

Theorem 9.13. *For every two positive integers n and l, there is m and ε_0 such that every n-CSP$_2$ instance f can be reduced to a 2-CSP$_m$ instance g in polynomial time, preserving satisfiability and multiplying the number of constraints by a bounded, constant factor, such that if $val(f) \leqslant 1-\varepsilon$, for $\varepsilon < \varepsilon_0$, then $val(g) \leqslant 1 - l\varepsilon$.*

Thus if ρ-GAP 2-CSP$_m$ is **NP** hard for all m, the above theorem shows that there is ν such that ν-GAP n-CSP$_2$ is **NP** hard, and thus the PCP theorem holds.

Håstad provides a verifier for the regular instances of 2-CSP$_n$ using only three queries, showing

The proof of Håstad's theorem, like the PCP lemma we proved earlier, requires a particular application of an error correction code to encode the data in a problem as a proof, in a way which maximizes the amount of data we obtain in a query. The **long code** on $[n]$ encodes each integer $m \in \{1,\dots,n\}$ as the dictator $x^m : \{-1,1\}^n \to \{-1,1\}$. This is a doubly exponential length encoding, since m is normally written with $\log n$ bits, but is now written with 2^{2^n} bits, but allows us to reduce checking properties of arbitrary $\log n$ bit strings into checking properties of dictators. By property testing, for any $\rho < 1$, given an arbitrary function $f : \mathbf{F}_2^n \to \mathbf{F}_2$, there is a verifier making only three queries to the bits of a function f, such that if f is a dictator, the algorithm accepts with probability $1 - \rho$, and if the test accepts with probability greater than $1/2 + \delta$, then

$$\sum \widehat{f}(S)^3 (1 - 2\rho)^{|S|} \geqslant 2\delta$$

hence f is very similar to a dictator. Indeed, an immediate corollary is that if $k = \log(1/\varepsilon)/2\rho$, there is S of cardinality less than or equal to k with $\widehat{f}(S) \geqslant 2\delta - \varepsilon$.

Now we describe Håstad's verifier. The goal of the verifier is, given two functions $f, g : \{0,1\}^n \to \{0,1\}$, and a function $h : \{1,\dots,n\} \to \{1,\dots,n\}$, to determine if f and g are the long codes of $x, y \in \{1,\dots,n\}$ respectively, and if $h(x) = y$. Define $H^{-1}(y)_k = y_{h(k)}$

Given a regular, 2-CSP f over the alphabet $\{1,\dots,m\}$ with the projection property, we expect a proof Π to consist of an assignment $\Pi_1,\dots,\Pi_n$ to the variables of f, where each Π_i is encoded using the long code as a dictator $\Pi_i : \mathbf{F}_2^m \to \mathbf{F}_2$. Given any proof Π, we can assume each Π_i is an odd function, by halving the length of the proof (since odd functions need only be specified on half of their input). Thus the proof has length $n 2^{2^m - 1}$

Another tool we shall have is a dictator test involving noise operations. Let $\rho > 0$ be arbitrary. Then pick $X, Y \in \{-1,1\}^n$ uniformly at random, and $Z \in \{-1,1\}^n$ is randomly chosen with independent coordinates such that $\mathbf{P}(Z_i = 1) = 1 - \rho$, $\mathbf{P}(Z_i = -1) = \rho$. We then test if $f(X)f(Y) = f(XYZ)$. If $f(x) = x_m$, then $X_m Y_m = X_m Y_m Z_m$ with probability $1 - \rho$. Note that this test only applies three queries to a certificate (the long code representation of f).

Theorem 9.14. *If the test passes on f with probability $1/2 + \delta$, then $\sum (1 - 2\rho)^{|S|} \widehat{f}(S)^3 \geqslant 2\delta$.*

Corollary 9.15. *If f passes the long code test with probability $1/2 + \delta$, then for $k = 1/(2\rho)\log(1/\varepsilon)$, there is S with $|S| \leqslant k$, and $\widehat{f}(S) \geqslant 2\delta - \varepsilon$.*

We now prove Hastad's theorem. Say a Boolean-valued function is **bifolded** if it is odd.

9.6 MAX-CUT Approximation Bounds and the Majority is Stablest Conjecture

Recall the MAX-CUT problem, which asks, given a graph G, to find a partition of the vertices of G into two sets V and W, such that the cardinality of the cut

$$\delta(V_0, V_1) = \{(v, w) \in E : v \in V, w \in W\}$$

is maximized. The problem is known to be NP-complete, and here we will derived tight approximation bounds for the problem. Using the techniques of semidefinite programming, one can derive an α-approximation algorithm, where

$$\alpha = \min_{0<t<\pi} \frac{2t}{\pi(1-\cos t)} \approx 0.878567$$

It may be surprising that geometry becomes involved in the MAX-CUT problem, but we will see the impact of geometry in our analysis that α is a tight constant approximation bound for the problem, assuming that the 'unique games conjecture' holds.

The unique games conjecture can be stated in terms of the inapproximability of a particular **NP** complete problem, known as the unique label cover problem. We are given a bipartite graph G with left vertices V and right vertices W, an alphabet Σ, and for each $(v, w) \in E$, a bijection $\pi_{(v,w)} : \Sigma \to \Sigma$. A labelling $L : (V \cup W) \to \Sigma$ then 'satisfies' an edge (v, w) if $\pi_{(v,w)}(L(v)) = L(w)$, and the goal of the unique label cover problem is to find a label which satisfies as many edges as possible. This problem can be formalized as a version of MAX-CSP, where each constraint, indexed by $(v, w) \in E$, is

$$f_{(v,w)}(L) = \text{``}\pi_{(v,w)}(L(v)) = L(w)\text{''}$$

which is

or alternatively, as a way to reduce any problem to a 2-query verifier

NP hardness result for

In order to obtain tight approximation bounds on the MAX-CUT problem, we introduce the techniques involves with the unique games conjecture, and its relation to the majority is stablest theorem.