A Modern Introduction to Logic and Computers

Subash Shankar

Contents

Chapter 1

Propositional Calculus (PC)

1.1 2000 Years of Logic

Logic is an interdisciplinary field, spanning philosophy, mathematics, linguistics, and computer science among other disciplines. It can be thought of as the foundational science that many of these disciplines are based on. It also has a long history, with a recent resurgence resulting from the burgeoning of ever-more-powerful computer technology. This book introduces the reader to logic and its applications, focusing on the automation of logical reasoning.

Early History

The story of logic as a formalizable system originates with the ancient Greeks, who formulated a deductive proof-oriented approach to mathematics. A proof of a theorem in the Euclidean geometry that we are all familiar with then becomes merely an application of fundamental logical arguments on the basic Euclidean postulates to deductively conclude the desired theorem in a justifiable manner. Although a number of Greek philosopher-mathematicians pondered various logical paradoxes and the use of rigorous logical reasoning to model the world, Aristotle is generally considered the father of logic for his formalization of logical reasoning in the 4th century BCE.

Aristotle is best known for the notion of a syllogism, where conclusions are drawn from premises based on their logical form[1]. For example, from the two premises:

All humans are mortal.
Socrates is human.

we may conclude that Socrates is mortal. The key is that this argument is not limited to humans/mortality/Socrates, instead generalizing to *any* argument of

[1] Aristotle's logic, like most of the approaches in this section, is actually for first order logic rather than propositional logic. It is described in this chapter on propositional logic since historical approaches to logic generally focused more on the more complex first order logic.

the form:

$$\begin{array}{ll} & \text{All S's are P.} \\ & \text{Object A is an S.} \\ \therefore & \text{A is P.} \end{array}$$

where $\therefore$ is read "therefore". Aristotle identified four such statements:

A	All S's are P.
E	No S is P.
I	Some S is P.
O	Some S is not P.

and showed rules to deductively infer various statements depending on the *form* of the given premises. Euclid's *Elements* demonstrated a formal system composed of axioms/postulates describing geometry along with rules similar to those of Aristotle, and proceeded to prove all theorems of geometry rigorously by explicitly identifying which axiom/rule was used for each step of the proof. This book will present various such systems.

It is interesting to recognize that Aristotle's formalization predates corresponding developments in other areas of mathematics by many centuries. In particular, a history of algebra may be partitioned into three stages:

1. Rhetorical algebra: Algebraic equations are written as natural language sentences – for example "the entity which when doubled is three more than itself". This was the dominant form for most of human history.

2. Syncopated algebra: An intermediate form that introduced abstract entities for some common forms, but not to the level of full symbolic algebra described below. Two prominent approaches were by the Greek mathematician Diophantus in the 3rd century and the Hindu mathematician Brahmagupta in the 7th century.

3. Symbolic algebra: This is the algebra that we are all familiar with today, where variables are abstract entities that may represent arbitrary values – for example $x * 2 = x + 3$. Preliminary versions of this appeared in the Arab world starting in the 13th century, with Vieta and Descartes introducing more complete versions in the 16th and 17th centuries. Its full adoption is even more recent.

Analogously, there is a large chronological gap from the [arguably] syncopated logic of Aristotle to the development of modern symbolic logic. The intervening period was largely a dark period for logic as mathematics and science were mostly empirical with only a little interest in formal reasoning. Nevertheless, some steps were taken towards the development of symbolic logic. The Franciscan monk Llull, influenced by earlier work by the Greeks and Arabs, designed mechanical devices to automate such reasoning in the 13th century. His work in turn inspired the mathematician Leibniz, who in the 1680s made progress toward the development of a formal language, *characteristica universalis*, that would

today be seen as propositional calculus, along with an automatable method, *calculus ratiocinator*, for performing reasoning in this language. Unfortunately, most of Leibniz's work was unpublished and not recognized until the early 1900s.

A Foundation for Mathematics

In 1847, Boole developed symbolic logic, and logic was quickly recognized as the foundation of mathematics and thus blossomed as a major subfield of mathematics. In the late 1800s, many philosophers and mathematicians including Cantor, Frege, and Hilbert worked to develop various logics and formalize much of mathematics in these logics, an approach called *logicism*. In particular, Cantor proposed and Frege further developed the *naive set theory*. The goal here was to provide a set of axioms for set theory from which all of mathematics could be derived, much in the same manner as Euclidean geometry could be derived from a small set of postulates. A successful axiomatization could then provide a more rigorous framework for the types of less formal reasoning done in other areas of mathematics such as algebra and calculus. This work also interacted with linguistics as it often involved the formalization of natural language statements.

Much of the earlier work in logic served to provide a formal language for expressing statements as well as formal rules for reasoning about such statements (and done well before the invention of the modern computer). In the early 1900s, logic research shifted focus to the theoretical properties of the logics themselves. Partly, this was driven by the identification of fundamental paradoxes – in particular, Russell's famous paradox showed that Frege's naive set theory was contradictory, leading to a foundational crisis in mathematics. The classic 3-volume *Principia Mathematica* by Russell and Whitehead in 1910-1913 presented a foundation of mathematics that included an attempt to resolve this paradox. Typical questions addressed by researchers in this period included the consistency of the logical systems themselves, and whether every logically valid statement is provable within the logical system. In particular, *Hilbert's Program* had the goal of providing a axiomatic foundation of mathematics ensuring positive answers to these questions, though Gödel later proved this was not possible (see Section 3.7). Additionally, if the formal reasoning can be mechanistically applied, computers can of course do it much faster (and correctly!). Indeed, much of computation theory was developed by the logicians Church, Turing, Gödel, Kleene, Post, Von Neumann and others as a way to understand the limits of computation, which apply to computers as well as humans. This theory was fundamental to the development of real computers in the 1940s.

A Foundation for Computer Science

Starting in the late 1950s, the goal of making machines that could emulate human thought (*i.e.*, artificial intelligence) saw a resurgence in logic since it seemed to be a good model of human thought. McCarthy, the person generally credited as the founder of artificial intelligence, expressed his opinion:

> The best hope for human-level AI is logical AI, based on the formalizing of commonsense knowledge and reasoning in mathematical logic.

In keeping with this, logic was used to 1) capture and represent knowledge, and 2) automate reasoning about knowledge. Logic is well matched to knowledge representation since logical predicates can express relationships between concepts or entities – for example, the predicate `mother(barb,bob)` captures the fact that barb is bob's mother. Simple forms of knowledge representation are essentially the core of relational database theory (*e.g.*, the mother relation above can also be considered a database fact). Knowledge representation has more recently become the major component of the semantic web, an attempt to organize knowledge available on the web into ontologies, thus allowing for web search engines that do more than simple keyword matching. In parallel to knowledge representation, automated reasoning led to the development of *theorem provers* that allowed for automated and semi-automated proof procedures. These theorem provers were used by mathematicians to prove formerly unproven theorems (*e.g.*, the famous four color theorem) as well as verify and/or correct the proofs of existing theorems, a few of which had been published several centuries prior!

Logic is also the major mathematical foundation of most computer science areas. Two in particular are hardware design and computational complexity. The former is evident since the basic logical operators also correspond to the fundamental building blocks of digital hardware, and manipulation of logical formulas is critical to efficient hardware design. In the area of computational complexity, it was proven that a class of many of the hardest computer science problems were all reducible to the problem of propositional logic satisfiability (to be defined in Section 1.5). This class of problems was called NP-complete, and good reasoning procedures for satisfiability thus may affect the efficiency of this huge class of problems in practice.

Going beyond the above foundational areas, computer software systems had also grown in size by the 1960s so that it was not feasible to verify that they actually worked correctly. The resulting foundational crisis (this time in computer science instead of mathematics) was coined as the *software crisis* in 1968, later expanding also to hardware as computers grew in power. The recognition of a software crisis led to numerous innovations including the development of programming language features and software development methodologies alleviating (but not solving) some issues – indeed most features of modern programming languages can be traced back to solutions derived to address the software crisis. However this still did not approach the goal of provably correct hardware and software systems, which is especially important with systems where bugs result in loss of life (safety critical systems) or large costs (*e.g.*, space based systems). This led to the rise of the *formal methods* area which developed the theory and approaches addressing the software crisis, along with its hardware analog. The primary goal of formal methods is to model computer systems in a formal language such as logic and apply automated reasoning techniques to prove that the systems work correctly. This contrasts with the prevalent

testing-based approach that merely checks correctness for a set of test cases, or as famously stated by Dijkstra, "to show the presence of bugs, but never their absence". Meanwhile in the late 1990s and 2000s, theorem provers and other related reasoning mechanisms had taken a huge leap due both to faster computers and improved algorithms, enabling the practical use of formal methods. Formal methods have been adopted in hardware today, and their application to software verification is a major research area. Related to formal methods, the areas of computer security and programming languages also use logic in similar ways (*e.g.*, to prove that a software system is secure, or to incorporate language features that support safer programs). Although this text is intended to give an introduction to logic in general, many of our applications focus on formal methods.

Onward

A major goal of logic is to formally and precisely represent statements, along with provably sound procedures to reason about such statements. Although our emphasis above has been on statements modeling mathematics and computer science areas, the statements being modeled by logic may be from other areas too, making logic a truly interdisciplinary foundational science. Ideally, the reasoning itself may be automated; however, there is a tradeoff between the expressive power of the logic and the ability to effectively automate reasoning in the logic. This chapter presents a very simple logic that sacrifices expressive power for powerful and simple reasoning techniques. This logic is called propositional calculus (also sentential logic, propositional logic) or PC for short, since it deals with the logic of true-false sentences/propositions. Despite its simplicity, many large classes of problems spanning areas including planning, scheduling, graph theory, and formal methods have successfully been modeled using PC. It is also the basis of most other logics, and the techniques covered here will be extended to these other logics in later chapters.

1.2 Syntax

To define a logic, it is important to distinguish the *form* of legal statements from the *meaning* of statements. The former is a purely syntactic construct, while the latter is referred to as semantics (the reader is cautioned that the common debate phrase, "That's just semantics" is often used incorrectly to refer to anything but semantics!). Consider for example, English statements. One form of English statement is a pronoun followed by a verb and then a noun. A common way of formally representing such syntactic forms is in Backus-Naur Form (BNF), where each such syntactic form is captured by a grammar rule. For example, the above is captured by:

$$< sentence > ::= < pronoun >< verb >< noun >$$

By instantiating each <>-delimited component, this rule can be instantiated to the declarative statements, "I eat pizza", "I write books", "I eat books", or "I

write dogs". The first 2 of these also make semantic sense, while the third is not meaningful except perhaps in a poetic sense (or if I am a goat) and the last is semantically nonsensical[2]. Nevertheless all four statements are syntactically legal – that is they can all be instantiated from the BNF Rule for this form of English sentence.

Def: A **well-formed formula**, or **wff** for short, is a syntactical legal statement in a formal language (such as propositional logic), where a statement is a finite sequence of symbols.

Syntax of the formal language is typically represented using a set of BNF rules, and each wff can be generated by finitely many applications of the rules. In general, only some subset of rules will be used to generate any particular wff – for example, the above English rule involved pronouns but many legal English sentences will not contain pronouns and thus not use that rule. Starting with a base set of true-false propositions, propositional logic defines operators that may be used to construct wffs.

Def: A **proposition**, also called **atom** or **propositional variable**, is a statement that can be true or false and is indivisible. In the abstract, it is typically denoted using a [possibly subscripted] lower-case letter, most often p, q, and r, though we may sometimes also use lower-case words for readability purposes[3].

Figure 1.1 lists some examples of propositions. Note that the actual truth/-falsity of these atoms is irrelevant, as one of them leads to deep theological discussions. It is not even required that the truth of the atom is determinable, as the last is a well known open problem. Also note the exclusion of statements that do not evaluate to true or false, some of which are listed in Figure 1.2. The last of these is particularly subtle as it is easy to confuse the statement that x is 2+3 (q_2 in Fig. 1.1) with the definition of x.

PC defines unary and binary operators that may be used to build wffs as follows:

- Atoms: Every atom is a wff.
- Constants (nullary operator): $\top$ and $\bot$ are wffs.
- Unary operators: If ϕ is a wff, $(\neg\phi)$ is a wff
- Binary operators: If ϕ and ψ are wffs, the following are all wffs:
  - $(\phi \wedge \psi)$ (conjunction)

[2]If the intent is to write a work titled "dogs" (*i.e.*, the mention of dogs instead of its use in the sentence), it would properly be written in a different font or surrounded by quotations. The distinction between *use* and *mention*, while important in linguistics and analytic philosophy is not the focus of this book.

[3]There is some inconsistency in the use of lower vs. upper case across logics and applications (and sources). In this book, we choose to be internally consistent where possible.

Atom	English Sentence
p_1	Barb likes logic
p_2	Logic is fun
$raining$	It is raining
q_1	2+3=6
q_2	x=2+3
fw	Humans have free will
pp	There are infinitely many palindromic primes

Figure 1.1: Some Sample Propositions

Non-proposition	Type
Does Barb like logic	Question
`x := x+1;`	Programming language statement
2+3	Number
Let x be 2+3	Definition

Figure 1.2: Some Sample Non-propositions (and why they aren't)

- $(\phi \vee \psi)$ (disjunction)
- $(\phi \rightarrow \psi)$ (implication)
- $(\phi \leftrightarrow \psi)$ (iff)

In the case of $(\phi \wedge \psi)$, ϕ and ψ are called **conjuncts**; similarly, they are called **disjuncts** for the $(\phi \vee \psi)$ case. For the implication, ϕ is called the **antecedent** and ψ is called the **consequent**.

The $\bot$ and $\top$ constants are read as false and true, respectively. The $\neg$, $\wedge$, $\vee$, $\rightarrow$, and $\leftrightarrow$ operators are read as not, and, or, implies, and if and only if, respectively[4]. The set of legal wffs ϕ may more compactly be represented in BNF as:

$$\phi ::= < atom > \mid \top \mid \bot \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \rightarrow \phi) \mid (\phi \leftrightarrow \phi)$$

where the $\mid$ symbol is shorthand to represent a choice between multiple rules, for example $\phi ::= < atom >$ and $\phi ::= \top$. The top-level operator used to construct the wff is called the **primary operator**. Each intermediate wff that occurs when building using these rules (along with ϕ itself) is called a **subformula**.

This BNF definition is an example of a recursive rule, where a syntactic entity (the wff ϕ here) is defined in terms of [smaller] versions of the same entity, and every legal entity is a result of finitely but arbitrarily many rule applications. This is common in grammars, and has the added advantage of

[4]For readability purposes, we are cheating by hinting at the semantics of the operators with these readings even though this section deals purely with syntax.

providing an effective method to prove a property of all wffs using structural induction.

Example 1.1: Suppose we wish to prove that all wffs have at least as many subformulas as operators (excluding constants). This is simple to prove using structural induction:

- Base cases (atoms and constants): Every atom or constant has only itself as subformula and no non-nullary operators. The result follows since 1>0.
- Inductive cases: We need to consider the unary and binary operator cases. Unary case: Then, $\phi \equiv \neg\phi_1$. Denote the number of subformulas of ϕ_1 as m, and let n be the number of operators in ϕ_1. Then, ϕ has $m+1$ subformulas and $n+1$ operators. Since $m \geq n$ by the inductive assumption, $m+1 \geq n+1$.
  Binary case: Then, $\phi \equiv \phi_1 * \phi_2$, where $* \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$. Denote the number of subformulas and operators in ϕ_1 as m_1 and n_1 respectively, and similarly for ϕ_2, m_2, and n_2. Then ϕ has $m_1 + m_2 + 1$ subformulas (those of ϕ_1 and ϕ_2 along with ϕ itself) and $n_1 + n_2 + 1$ operators. Since $m_1 \geq n_1$ and $m_2 \geq n_2$, it follows that $m_1 + m_2 + 1 \geq n_1 + n_2 + 1$.
  Thus, both the unary and binary inductive case hold.

By structural induction, we have shown the desired statement.

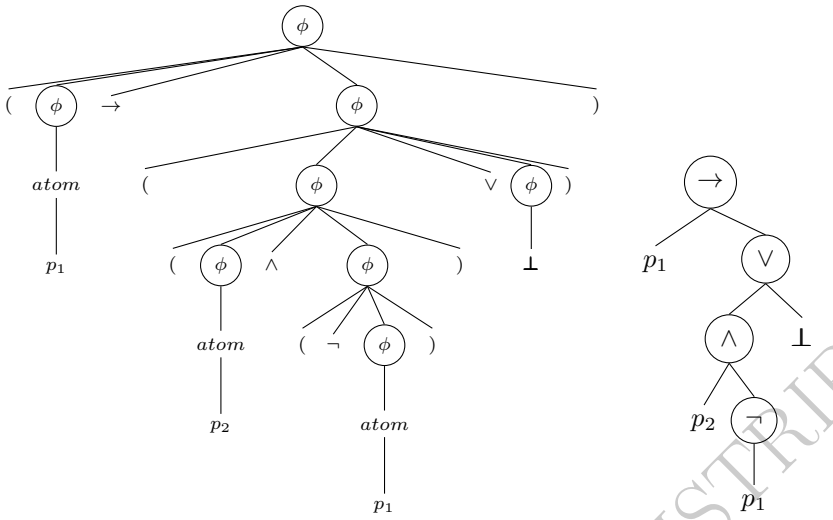
Note that the the power of induction allows us to prove the above property of wffs for arbitrarily large wffs even though the proof itself need only cover cases corresponding to the BNF grammar definition. Note also that there are infinitely many such wffs, though the proof need only cover a small number of cases.

The BNF definition also provides a way to check that a candidate string of symbols is actually a wff:

Example 1.2: Consider the string $(\neg(p \wedge q))$. Both p and q are wffs since they are atoms. By the 'and' rule, $(p \wedge q)$ is a wff. By the 'not' rule, the string is indeed a wff.

The subformulas are p, q, $(p \wedge q)$, and $(\neg(p \wedge q))$. Its primary operator is $\neg$.

The above argument to show that a candidate wff is a wff becomes unwieldy for larger wffs. An alternate representation is as a **parse tree** that shows how the wff can be parsed using the BNF rule. Figure 1.3(a) illustrates such a parse tree (also called concrete syntax tree), where leaf nodes (*i.e.*, those without children) correspond to atoms, constants, and punctuation symbols, and non-leaf nodes correspond to wffs that need expansion in accordance with a BNF rule (*i.e.*, an application of a unary or binary PC operator). An infix left-to-right traversal through leaves of the parse tree results in the formula being parsed. Since the size of parse trees can be unwieldy, an abstracted form



(a) Parse Tree (b) Formation Tree

Figure 1.3: Syntax Trees for $(p_1 \to ((p_2 \wedge (\neg p_1)) \vee \bot))$

without nodes corresponding to syntactic entities such as parentheses is more often used (Figure 1.3(b)). This tree is called a **formation tree**, also called abstract syntax tree, with the former term more common in the logic world. Thus, each subformula of a wff corresponds to some subtree of its formation tree.

Avoiding Parentheses The given BNF grammar generates wffs with a profusion of parentheses, leading to readability issues. To alleviate this problem, we impose a precedence order on PC operators so that the unary operator $\neg$ is always higher priority than binary operators[5]. For example, the wff $(\neg p \vee q)$ is intended shorthand for $((\neg p) \vee q)$, not $(\neg(p \vee q))$. In addition, we typically omit parentheses in wffs when it does not lead to semantic ambiguity. In particular, parentheses around $\neg\phi$ and outermost parentheses are often omitted. For example, $(((\neg p_1) \wedge p_2) \vee q)$ may be written as $(\neg p_1 \wedge p_2) \vee q$. But it may **not** be written as $\neg p_1 \wedge p_2 \vee q$, since that may also refer to the wff $((\neg p_1) \wedge (p_2 \vee q))$ and these two wffs will be shown in the next section to have different semantics. Similarly, parentheses are always needed when connecting three subformulas using *different* operators, for example in $(p \to q) \to r$ or $p \to (q \vee r)$, which would be semantically ambiguous otherwise.

[5]Some sources also impose a precedence order between binary operators, but the order may differ across sources and we opt against this for simplicity, though at the cost of extra parentheses.

1.3 Semantics

The semantics of a formal language gives meaning to a wff by mapping it to some precise formalism or language. The most common such formalisms are:

- Model theoretic: mathematical structures representing the truth and falsity of a wff. It was originally introduced by the mathematician Tarski in 1954, and is now a major subfield in the foundations of mathematics.

- Proof theoretic: rules and/or procedures that may be used to prove the truth of a wff. This is traditionally done using axioms from which a wff can be derived, much as Euclid's axioms can be used to prove geometry. In this book, we loosely use algorithms/programs in addition since our goal is automation.

- Game theoretic: rules for a game between players (*e.g.*, a truthifier and a falsifier), such that the truth of the wff is determined by the game's winner. This form of semantics was originally mentioned in the 1890s by Peirce and is sometimes called Hintikka semantics, eponymously named for its primary developer. Although less common than the above forms, it is well matched to some logics and applications.

Each of these may itself have multiple techniques – in particular, there are many different proof procedures, and the major classes of these will be discussed in later sections of this chapter, after first discussing model theoretic and game theoretic semantics.

Model Theoretic Semantics of PC

Model theoretic semantics was introduced by the logician Tarski to map the meaning of wffs in a logic to mathematical domains. In the case of propositional logic, we start by capturing the truth/falsity of each atom:

Def: Given a set of atoms $\mathcal{A}$, a **model** $\mathcal{M}$ is a valuation function $\eta : \mathcal{A} \rightarrow \{\bot, \top\}$ assigning either $\top$ or $\bot$ to each atom in $\mathcal{A}$. It is sometimes more compact to represent η by the set of all atoms true in $\mathcal{M}$ (and no others), which is denoted $\mathcal{L}_{\mathcal{M}}$. If a wff ϕ holds in $\mathcal{M}$, it is denoted as:

$$\mathcal{M} \models \phi$$

The $\models$ symbol is read as [the verb] "models" or "double turnstile" (to distinguish from the single turnstile that will later be used for proof theoretic semantics), and the above is thus read as "$\mathcal{M}$ models ϕ" or "$\mathcal{M}$ is a model of ϕ".

Example 1.3: Let $\mathcal{A} = \{p_1, p_2, p_3\}$.

One model $\mathcal{M}_1$ over this atom set assigns T, $\bot$, and $\bot$ to p_1, p_2, and p_3 respectively, or equivalently $\mathcal{L}_{\mathcal{M}_1} = \{p_1\}$. $\mathcal{M}_1 \models p_1$ but $\mathcal{M}_1 \not\models p_2$ clearly.

Another model $\mathcal{M}_2$ assigns $\bot$ to all atoms in $\mathcal{A}$, or $\mathcal{L}_{\mathcal{M}_2} = \{\}$. Clearly, $\mathcal{M}_2 \not\models p_1$ and $\mathcal{M}_2 \not\models p_2$.

Since there are 3 atoms, there are 2^3=8 total models, 4 of which model p_1.

Given a model $\mathcal{M}$ and a wff ϕ with atoms in $\mathcal{A}$, the model theoretic semantics of the wff can be recursively defined based on the syntax of the wff:

1. For any atom $a \in \mathcal{A}$, $\mathcal{M} \models a$ iff $\eta(a) = \mathsf{T}$
2. $\mathcal{M} \models \mathsf{T}$
3. $\mathcal{M} \not\models \bot$
4. $\mathcal{M} \models (\neg\phi)$ iff $\mathcal{M} \not\models \phi$
5. $\mathcal{M} \models (\phi \wedge \psi)$ iff $\mathcal{M} \models \phi$ and $\mathcal{M} \models \psi$
6. $\mathcal{M} \models (\phi \vee \psi)$ iff $\mathcal{M} \models \phi$ or $\mathcal{M} \models \psi$
7. $\mathcal{M} \models (\phi \rightarrow \psi)$ iff $\mathcal{M} \models \neg\phi$ or $\mathcal{M} \models \psi$
8. $\mathcal{M} \models (\phi \leftrightarrow \psi)$ iff $\mathcal{M} \models \phi \rightarrow \psi$ and $\mathcal{M} \models \psi \rightarrow \phi$

From the semantics, it is simple to evaluate the truth of a wff in a model:

Example 1.4: Consider the wff $\phi \equiv \neg p_1 \rightarrow (p_2 \wedge \neg p_3)$ and the model $\mathcal{M}$ corresponding to $\mathcal{L}_{\mathcal{M}} = \{p_2\}$. To show that $\mathcal{M} \models \phi$ using the model theoretic semantics:

$$
\begin{aligned}
\mathcal{M} \models \phi \text{ iff } & \mathcal{M} \models \neg\neg p_1 \text{ or } (\mathcal{M} \models p_2 \wedge \neg p_3) \\
\text{iff } & \mathcal{M} \not\models \neg p_1 \text{ or } (\mathcal{M} \models p_2 \wedge \neg p_3) \\
\text{iff } & \mathcal{M} \models p_1 \text{ or } (\mathcal{M} \models p_2 \wedge \neg p_3) \\
\text{iff } & \mathcal{M} \models p_1 \text{ or } (\mathcal{M} \models p_2 \text{ and } \mathcal{M} \models \neg p_3) \\
\text{iff } & \mathcal{M} \models p_2 \text{ and } \mathcal{M} \models \neg p_3 \text{ (since } p_1 \notin \mathcal{L}_{\mathcal{M}}) \\
\text{iff } & \mathcal{M} \models \neg p_3 \text{ (since } p_2 \in \mathcal{L}_{\mathcal{M}}) \\
\text{iff } & \mathcal{M} \not\models p_3
\end{aligned}
$$

which is true since $p_3 \notin \mathcal{L}_{\mathcal{M}}$

There are several interesting issues in the previous definition of a model. First, the use of the words "and" and "or" are in the meta-language, not the

object language where the $\wedge$ and $\vee$ operators appear (even though they may be read the same way). Thus, our use of these meta-language terms in the above example is justified, and not a circular definition.

Also, the logical operators do not always correspond to the common natural language interpretation, with two particularly important distinctions. First, the $\vee$ operator refers to the inclusive rather than exclusive or. For example, the English statement "you will study or you will fail" generally precludes the possibility of both studying and failing, while the inclusive or in logic does not. Second, the $\rightarrow$ operator refers to material implication where no connection is needed between its operands, unlike in English. For example, "if I eat pizza, I get an A" would not be a typical English statement since the act of eating pizza is irrelevant to the grade, but the wff $eatpizza \rightarrow getA$ holds as long as either I don't eat pizza or I get an A. In the first case where I do not eat pizza, the wff is said to hold vacuously (regardless of my grade).

We often find it useful to extend the definition of $\models$ as follows:

Def: Given wffs $\{\phi_i\}_{i=1\ldots n}$ and $\{\psi_i\}_{i=1\ldots m}$, the 'models' relation is extended so that $\phi_1, \phi_2, \ldots, \phi_n \models \psi_1, \psi_2, \ldots, \psi_m$ iff for every model $\mathcal{M}$ of $\phi_1 \wedge \phi_2 \wedge \ldots \wedge \phi_n$, $\mathcal{M} \models \psi_1 \vee \psi_2 \ldots \vee \psi_m$.
This is often read as $\phi_1, \phi_2, \ldots \phi_n$ **entails** $\psi_1, \psi_2, \ldots, \psi_m$ (as well as "models" like before).

Here, the two sides of the $\models$ symbol are collections of wffs instead of a simple valuation for the left side or wff for the right side. Indeed, the form $\phi_1, \phi_2, \ldots, \phi_n \models \psi$ is the more common use of $\models$ in PC, though not in some other logics discussed later.

The model theoretic semantics also reveals a correspondence between set theory and logic. Let S_1 and S_2 be the sets of models of ϕ_1 and ϕ_2, respectively. Then, $S_1 \cup S_2$ is the set of models of $\phi_1 \vee \phi_2$, and $S_1 \cap S_2$ is the set of models of $\phi_1 \wedge \phi_2$[6]. If $\phi_1 \rightarrow \phi_2$, this means that every model of ϕ_1 is also a model of ϕ_2, which corresponds to $S_1 \subseteq S_2$. If $\phi_1 \leftrightarrow \phi_2$, the models of ϕ_1 are exactly the same as those of ϕ_2. This correspondence between set theory and logic can be exploited to prove any PC identities by instead using set theory identities, or vice versa.

1.4 PC as a Modeling Language

Despite the seeming similarity between logical operators and English words, it is not always simple to formalize an English sentence in propositional logic. Given a sentence, we first need to identify the propositions in the sentence, and then express the sentence using PC operators over these propositions. We start with an example:

[6]The reader might observe the visual similarity of corresponding symbols in set theory and logic.

If it is raining and I am outside, I get wet.
But if I am wearing a raincoat, I do not get wet.
If it is sunny, I do not get wet.

A meaningful formalization of this clearly involves 4 propositions:

rain	It is raining
outside	I am outside
wet	I get wet
raincoat	I am wearing a raincoat

We have chosen not to include a *sunny* atom, since this is better captured as $\neg rain$ as long as we are satisfied to exclude the possibility of a world in which it can be raining while also being sunny. The first sentence is simple to formalize as:

$$(rain \wedge outside) \rightarrow wet$$

if it were in isolation. Note that it does not say that being outside in the rain is the only way to get wet, as wading through a pond may be an alternate mechanism. The second sentence needs further thought since it implicitly refers to the first sentence, and the "but" indicates that this is an exception to the first sentence. A proper formalization needs to make this exception explicit, and one way is to replace the previous wff with two wffs:

$$((rain \wedge outside) \wedge \neg raincoat) \rightarrow wet$$

$$raincoat \rightarrow \neg wet$$

Both wffs are needed, as leaving out the second wff would allow for getting wet despite wearing a raincoat, perhaps by wading through a lake. The third sentence is:

$$\neg rain \rightarrow \neg wet$$

The formalization of the sentences is then the conjunction of the three wffs given above:

$$(((rain \wedge outside) \wedge \neg raincoat) \rightarrow wet) \wedge (raincoat \rightarrow \neg wet) \wedge (\neg rain \rightarrow \neg wet)$$

There are several words commonly used in English that map to PC operators, and we now turn to discussing these. The reader is warned that while other natural languages also have such issues, they may not always be handled in the same manner.

Conjunction The word "and" obviously maps to $\wedge$, but other English words connecting English clauses also map to $\wedge$, and there is an implicit $\wedge$ connecting English sentences. The connection may be hidden or use words other than "and". For example, the following sentences all map to $rain \wedge outside$:

It is raining but I am outside.
Although it is raining, I am outside.
I am outside even though it is raining.

Note that the uses of words like "but", "although", and "even though" in these sentences imply additional implied causalities, and these are not captured in the formalization. A more complicated formalization might also capture these causalities using additional atoms.

English sentences with multiple concepts can also be captured using $\wedge$:

> It is a warm rainy day.

This may be best formalized as $warm \wedge rainy$, even though there is no word explicitly corresponding to the conjunction.

Regardless of which English word (or none) a particular $\wedge$ is meant to capture, all uses of the English "and" do not necessarily correspond exactly to logical conjunction. For example, consider the sentences:

> I ate and am stuffed.
> I got into the car and drove home.

A simple conjunction does not seem to capture the totality of these sentences. The first sentence implicitly has an additional causality between the two conjuncts, while the second sentence implicitly has an additional temporal relation between the two conjuncts. Such issues need to be considered in light of the intended goals of the formalization, and PC formalizations generally will not cover all vagaries.

Disjunction For the $\vee$ operator, the primary complication arises from the distinction between the inclusive and exclusive or discussed earlier. Here, it is important to analyze the English sentence to determine which is meant. For example, consider the sentences:

> Bob's pet is a dog or cat.
> Bob has a dog or cat.
> Bob has either a dog or cat.
> Bob has neither a dog nor a cat.

The first one is best formulated as an exclusive or, while the second one may arguably be an inclusive or, not precluding owning both a dog and cat. The third sentence emphasizes that the exclusive or is intended. The final sentence may best be formalized as $\neg dog \wedge \neg cat$, or $\neg(dog \vee cat)$, which is the negation of an inclusive or.

Negation While it is obvious that "not" and its many variations correspond to negation, negation can still be tricky to capture since care is needed to identify exactly which word/clause is being negated. The formalization of an English statement sometimes reveals bad grammar, and this is accentuated in the presence of negatives. For example, the sentence "The building will not have hot and cold water today" is properly formalized as $\neg(hot \wedge cold)$, since the rules of English grammar dictate that the "not" applies to the entire predicate However, it is probably intended to mean $\neg hot \wedge \neg cold$ which would be properly stated

"The building will have neither hot nor cold water today". These issues reflect common grammatical errors, but their resolution and formalization may need contextual knowledge.

Implication Formalizations mapping to the $\rightarrow$ operator often need more thought. We have already discussed the issues concerning the material implication meaning of $\rightarrow$, namely that there need not be any conceptual connection between its operands, and that an implication is vacuously true if the left side is false. The mapping for a sentence with the word "if" is normally simple, and the following sentences are all formalized as the wff $rain \rightarrow wet$ (note that the "only if" in the third sentence changes the direction of the implication):

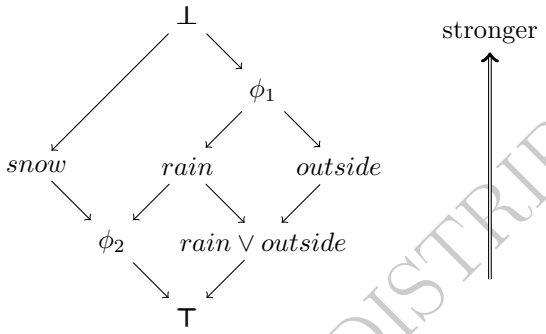
If it is raining, I get wet.
I get wet if it is raining.
It is raining only if I get wet.

However, many implications do not have an explicit "if"; for example, the sentence "rain makes me wet" may be formalized identically.

Another use of implications is in representing the concepts of **necessary** and **sufficient** conditions, which are commonplace in mathematical statements. Consider the statement, "it is sufficient to study to pass". The appropriate interpretation is that in every model in which a person studies, the person also passes, or $study \rightarrow pass$. Conversely, the statement "it is necessary to study to pass" does not guarantee a passing grade if a person studies; it only states that anybody who passes must have studied, or $pass \rightarrow study$.

Yet another interpretation of material implication is based on the observation that if $\phi_1 \rightarrow \phi_2$, ϕ_1 is **stronger** than ϕ_2, or equivalently that ϕ_2 is **weaker** than ϕ_1. Thus, implications may also be used to represent a hierarchy of stronger-to-weaker statements. The strongest wff of all (for any $\mathcal{A}$) is then $\bot$ since $\bot \rightarrow \phi$ vacuously for any wff ϕ. Similarly, the weakest wff is $\top$ since $\phi \rightarrow \top$ for any ϕ, even $\phi = \bot$.

Example 1.5: Let $\phi_1 \equiv rain \wedge outside$, and $\phi_2 \equiv rain \vee snow$. Then, $\phi_1 \rightarrow rain$, and ϕ_1 is clearly a stronger statement than $rain$.
Also, $rain \rightarrow \phi_2$, and ϕ_2 is weaker than just $rain$. These implications (along with a few others) can be arranged in an ordering to better illustrate the strength hierarchy:



Note that some wffs such as $snow$ and ϕ_1 may be incomparable in strength.

Since a stronger statement implies a weaker statement, the weaker statement also has at least as many models as the stronger statement, and the strength hierarchy can also be viewed as a hierarchy of sets of models[7].

Iff The $\leftrightarrow$ operator is not commonly used in normal prose. However its semantics are the same as equality, and it can thus be used to formalize any English statements referring to two propositions that hold in the exact same models or in definitional ways. For example, "A human is an erect primate" would seem to be definitional in nature and thus formalized as $human \leftrightarrow (erect \wedge primate)$. But if the context of the statement is to give characteristics of humans, a better formalization might be $human \rightarrow (erect \wedge primate)$.

Converse, Inverse, and Contrapositive Many mathematical statements and theorems are implications, and several similar forms are thus of interest, especially in mathematical arguments. Given an implication of the form $\phi \rightarrow \psi$,

- The **converse** of $\phi \rightarrow \psi$ is $\psi \rightarrow \phi$.
- The **inverse** of $\phi \rightarrow \psi$ is $\neg\phi \rightarrow \neg\psi$.
- The **contrapositive** of $\phi \rightarrow \psi$ is $\neg\psi \rightarrow \neg\phi$.

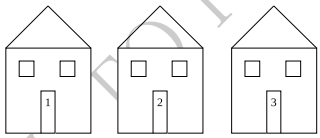
[7] More precisely, this is a lattice where the elements are sets of models over some fixed $\mathcal{A}$ set and ordered by set inclusion.

It is simple to show using model theoretic semantics that the contrapositive of a statement is equivalent to the statement. Similarly, the inverse and converse of a statement are equivalent. However, the converse of a statement is not equivalent to the original statement – indeed, the confusion of a statement with its converse is a common logical fallacy (called *fallacy of the converse*, *converse error*, or *affirming the consequent*). The contrapositive form is particularly useful since many mathematical theorems are implications, and it is their contrapositives that are often applied.

Our definition of PC syntax implicitly requires that all PC wffs are finite – that is, they result from finitely many applications of BNF rules, and can thus only have finitely many operators. This finiteness restriction forms one reason why all statements may not be expressible in PC. For example, an attempted formalization of "Everybody is mortal." might be $mortal_1 \wedge mortal_2 \wedge \ldots$ where the $mortal_k$ atom signifies that person k is mortal, but this is not a legal wff. This approach would work in concept if the population can be bounded to some finite set, though the size of the wff might then be impractical. We could have of course made this (or any other) entire statement a single atom, but that would be of little value though technically correct.

We close this section with an example of a puzzle that we will formalize now and solve later:

Example 1.6: There are 3 houses in a row called H1, H2, and H3, and they are colored blue, red, and white (in any order).



The 3 residents of the houses (1 per house) are a math major, a computer science major, and a philosophy major. We are also given the following facts:

1. The philosophy major lives directly to the right of the red house.
2. The computer science major lives in the blue house.
3. The math major lives in house 2.

We wish to determine for each house, its color and its resident, which we will do by expressing the puzzle in PC and later solving it.

We first identify the atoms. To do this, we need to characterize all possible models of the system, which are essentially the colors of and majors residing in each house.

p1	H1 is blue	p2	H2 is blue	p3	H3 is blue
p4	H1 is red	p5	H2 is red	p6	H3 is red
p7	H1 is white	p8	H2 is white	p9	H3 is white
p10	H1 has math major	p11	H2 has math major	p12	H3 has math major
p13	H1 has CS major	p14	H2 has CS major	p15	H3 has CS major
p16	H1 has phil major	p17	H2 has phil major	p18	H3 has phil major

We have made a choice here in not selecting an independent set of atoms since some atoms may be expressed in terms of others – *e.g.*, $p3$ could be written as $\neg p1 \wedge \neg p2$. However we choose to have extra atoms with explicit wffs corresponding to the constraints relating these. Had we chosen a completely independent set of atoms, these additional constraints would not be needed thus simplifying the formalization below. The choice here however allows for more uniformity of wffs in the formalization.

To formalize the problem itself, we require constraints corresponding to the general setup as well as those for the 3 given facts. For the general setup, the conjunction of the following constraints are needed:

- Every house has a color:
  $p1 \vee p4 \vee p7$ $p2 \vee p5 \vee p8$ $p3 \vee p6 \vee p9$
  This statement does not preclude a house from having more than one color, and it is in fact common to formalize such constraints in two parts stating the minimum and maximum cardinality constraints (both 1 in this case).

- A house can not have more than one color (the maximum cardinality counterparts to the above):
  $p1 \rightarrow \neg p4$ $p1 \rightarrow \neg p7$ $p4 \rightarrow \neg p7$
  $p2 \rightarrow \neg p5$ $p2 \rightarrow \neg p8$ $p5 \rightarrow \neg p8$
  $p3 \rightarrow \neg p6$ $p3 \rightarrow \neg p9$ $p6 \rightarrow \neg p9$
  Note that while it might be tempting to formalize the first line of wffs more compactly as $\neg p1 \vee \neg p4 \vee \neg p7$, that would be incorrect since it allows for the models where house 1 has 2 but not 3 colors.
  Also note the exclusion of some wffs when their contrapositive form is present – for example, there is no need to state that $p7 \rightarrow \neg p1$ since its contrapositive $p1 \rightarrow \neg p7$ is present.

- A color can not be on more than one house:
  $p1 \rightarrow \neg p2$ $p1 \rightarrow \neg p3$ $p2 \rightarrow \neg p3$
  $p4 \rightarrow \neg p5$ $p4 \rightarrow \neg p6$ $p5 \rightarrow \neg p6$
  $p7 \rightarrow \neg p8$ $p7 \rightarrow \neg p9$ $p8 \rightarrow \neg p9$

- Every house has a resident:
  $p10 \vee p13 \vee p16$ $p11 \vee p14 \vee p17$ $p12 \vee p15 \vee p18$

- A house can not have more than one resident:
  $p10 \rightarrow \neg p13$ $p10 \rightarrow \neg p16$ $p13 \rightarrow \neg p16$
  $p11 \rightarrow \neg p14$ $p11 \rightarrow \neg p17$ $p14 \rightarrow \neg p17$
  $p12 \rightarrow \neg p15$ $p12 \rightarrow \neg p18$ $p15 \rightarrow \neg p18$

- A resident can not be in more than one house:
  $p10 \rightarrow \neg p11$ $p10 \rightarrow \neg p12$ $p11 \rightarrow \neg p12$
  $p13 \rightarrow \neg p14$ $p13 \rightarrow \neg p15$ $p14 \rightarrow \neg p15$
  $p16 \rightarrow \neg p17$ $p16 \rightarrow \neg p18$ $p17 \rightarrow \neg p18$

We have taken care to formalize only the statement of the puzzle, and not any implicit reasoning that we could have done to simplify the puzzle – after all, our goal is to formalize the puzzle, not solve it! By respecting this, we have also avoided inadvertently introducing flaws in logical reasoning. Finally, we are ready to formalize the given problem itself:

- The philosophy major lives directly to the right of the red house:
  $p4 \rightarrow p17$ $p5 \rightarrow p18$ $\neg p6$
- The computer science major lives in the blue house:
  $p1 \rightarrow p13$ $p2 \rightarrow p14$ $p3 \rightarrow p15$
- The math major lives in house two:
  $p11$

As can be seen, the encoding of even a simple problem is often quite lengthy in order to capture all constraints implicit to the English statement of the problem!

1.5 Three Fundamental Problems: SAT, VAL, and MC

The $\models$ relation forms the basis for all three fundamental problems in logic when used as a modeling language:

- **Satisfiability (SAT)**: Given a wff ϕ, does there exist a model $\mathcal{M}$ such that $\mathcal{M} \models \phi$, which is alternatively read "$\mathcal{M}$ satisfies ϕ" or "ϕ is satisfied by $\mathcal{M}$ (along with "$\mathcal{M}$ models ϕ" as before). This is a decision problem (*i.e.*, it has a yes/no answer), but a related and often more useful problem is to actually determine a $\mathcal{M}$ satisfying ϕ and it is common (though imprecise) to also refer to that as the SAT problem. However, any theoretical references to the SAT problem refer to only the decision problem.
- **Validity (VAL)**: Given a wff ϕ does $\mathcal{M} \models \phi$ for every model $\mathcal{M}$? If ϕ is valid, it is denoted $\models \phi$. In some logics (including PC) ϕ is then called a **tautology** (or tautologically valid).
- **Model Checking (MC)**: Given a wff ϕ and a model $\mathcal{M}$, does $\mathcal{M} \models \phi$?

It is important to distinguish unsatisfiability and invalidity. A wff ϕ is **invalid** if it is not valid – that is, there is *some* model $\mathcal{M}$ such that $\mathcal{M} \not\models \phi$. Equivalently, a valid wff holds regardless of the truth value of atoms in the wff, while a wff is invalid if there is some truth assignment to the atoms under which it does not hold. A wff ϕ is said to be **unsatisfiable** if there is no $\mathcal{M}$ such that $\mathcal{M} \models \phi$, or equivalently $\mathcal{M} \not\models \phi$ for *every* model $\mathcal{M}$. Thus, the cases where ϕ hold in no and all models correspond to unsatisfiability and validity respectively. But ϕ is both satisfiable and invalid if it holds in some models but not others.

Example 1.7:
The wff $p \vee q$ is satisfiable but invalid, as it is satisfied in the model $\{p\}$ but not in the model $\{\}$ (recall that this latter model is defined to assign $\perp$ too all atoms).

The wff $p \vee \neg p$ is both satisfiable and valid, as it is satisfied regardless of what value a model assigns to p.

An example of an unsatisfiable wff is $p \wedge \neg p$.

All of these can be proven by enumerating models, and the rest of this chapter will present automatable techniques for showing such results.

The applications of the three problems are apparent if a model in the logical sense is viewed as the more common English notion of a model as a representation of a real-world system. In that case, SAT is simply determining a system satisfying a property represented as a logical wff – for example, determining the solution to a logical puzzle or game. Similarly, VAL is related to the problem of proving that a property ϕ must always hold – for example, showing that a statement is an algebraic theorem.

The MC problem is typically used only in logics with more expressive models than in PC, but it is useful to show that a particular system/model satisfies a certain property – for example, that [the formalization of] the software controlling an airplane does not cause a crash. In practice, a system corresponding to exactly one model is of limited utility as most real systems have multiple models – for example, a system with inputs $x_1, \ldots, x_n$ might correspond to a different model for each combination of values for the x_i's. Thus, the MC problem is often extended to:

- **Model Checking (MC)**: Given a wff ϕ and a *set of* models $\{\mathcal{M}_1, \ldots, \mathcal{M}_n\}$, does $\mathcal{M}_k \models \phi$ for every $1 \leq k \leq n$?

The reader may notice that the difference between these two MC versions parallels the difference between the definition of $\models$ and its extension to entailment (in Section 1.3). If the set of models is the universal set of all possible models, this problem is then just the validity problem; however, the number of models is in practice far smaller than the total number of models, which is 2^k if $|\mathcal{A}| = k$, and it is thus useful to treat the VAL problem separately. As is common, we will freely refer to this extended version as the MC problem going forward.

The SAT and VAL problems can also be related to each other:

Theorem 1.1 ***(Unsatisfiability Theorem)****: A PC wff ϕ is valid iff $\neg\phi$ is unsatisfiable.*

Proof:
$\Rightarrow$: Let $\mathcal{M}$ be any model and suppose ϕ is valid. Thus $\mathcal{M} \models \phi$. The model theoretic semantics then guarantees that $\mathcal{M}$ does not model $\neg\phi$. Thus $\neg\phi$ is unsatisfiable.

$\Leftarrow$: Suppose $\neg\phi$ is unsatisfiable. Then for every model $\mathcal{M}$, $\mathcal{M} \not\models \neg\phi$. By the model theoretic semantics, $\mathcal{M} \models \phi$ (for every model), which is the same as saying ϕ is valid.
QED

This theorem will be routinely used in later sections since it is common to solve the validity problem as a [un-] satisfiability problem instead.

The automation of these problems for various different logics is a major focus of logic today. To no surprise, there is a tradeoff between the expressiveness of the logic and the automatability of these problems.

1.6 Truth Tables for SAT, VAL, MC

Recall that the proof theoretic semantics of a logic provides rules and/or procedures for checking the validity of a wff. For logics with a finite number of models (such as PC), truth tables form a very simple procedure that can be used for all 3 fundamental problems. We discuss the truth table approach in this section, and some other approaches later.

Consider a wff ϕ with n atoms. There are 2^n models, that may be enumerated in tabular form, one model per row and one column per subformula. By convention, rows are ordered lexicographically (*i.e.*, dictionary order) where $\bot < \top$. The truth table is filled based on the operator semantics presented earlier.

Example 1.8: Consider the wff $\phi \equiv (p \wedge (p \rightarrow q)) \rightarrow q$. The truth table contains $4{=}2^2$ rows (since there are 2 atoms), and 5 columns since there are 5 subformulas (including ϕ).

p	q	$p \rightarrow q$	$p \wedge (p \rightarrow q)$	ϕ
$\bot$	$\bot$	$\top$	$\bot$	$\top$
$\bot$	$\top$	$\top$	$\bot$	$\top$
$\top$	$\bot$	$\bot$	$\bot$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

If there is some row of the resulting table in which ϕ holds, the corresponding model is a solution to SAT. If ϕ holds in every row of the table, ϕ is valid (as is true in Example 1.8). For the MC problem, the model/s are given and only the corresponding row/s of the truth table need be constructed.

Although truth tables are simple and sufficient to solve all three problems in principle, they quickly become unwieldy as the number of atoms grows – for example, a 20 atom wff would require over a million rows, and a 300 atom wff would require more rows than there are atoms in the universe (estimated to be 10^{80})! This motivates the search for other PC proof procedures, which we discuss after some preliminaries.

1.7 PC Identities and Normal Forms

Our PC syntax is redundant, as many operators can be expressed in terms of others. Two of these are:

$$\phi_1 \rightarrow \phi_2 = \neg\phi_1 \vee \phi_2$$
$$\phi_1 \leftrightarrow \phi_2 = (\neg\phi_1 \vee \phi_2) \wedge (\neg\phi_2 \vee \phi_1)$$

These identities can be used to replace any $\rightarrow$ and $\leftrightarrow$ symbols in a wff with the three basic symbols: $\neg$, $\wedge$, and $\vee$. Similarly, other identities can be used to eliminate constants. There are various useful identities for manipulating PC wffs (after $\rightarrow$ and $\leftrightarrow$ have been eliminated), as listed in Figure 1.4. It is of

Constant	$\neg\bot = \top$ $\neg\top = \bot$
Negation	$\phi \wedge \neg\phi = \bot$ $\phi \vee \neg\phi = \top$
Double Negation	$\neg\neg\phi = \phi$
Commutativity	$\phi_1 \wedge \phi_2 = \phi_2 \wedge \phi_1$ $\phi_1 \vee \phi_2 = \phi_2 \vee \phi_1$
Associativity	$(\phi_1 \wedge \phi_2) \wedge \phi_3 = \phi_1 \wedge (\phi_2 \wedge \phi_3)$ $(\phi_1 \vee \phi_2) \vee \phi_3 = \phi_1 \vee (\phi_2 \vee \phi_3)$
Distributivity	$\phi_1 \wedge (\phi_2 \vee \phi_3) = (\phi_1 \wedge \phi_2) \vee (\phi_1 \wedge \phi_3)$ $\phi_1 \vee (\phi_2 \wedge \phi_3) = (\phi_1 \vee \phi_2) \wedge (\phi_1 \vee \phi_3)$
Duality (De Morgan)	$\neg(\phi_1 \wedge \phi_2) = \neg\phi_1 \vee \neg\phi_2$ $\neg(\phi_1 \vee \phi_2) = \neg\phi_1 \wedge \neg\phi_2$
Identity	$\phi \wedge \top = \phi$ $\phi \vee \bot = \phi$
Annulment	$\phi \wedge \bot = \bot$ $\phi \vee \top = \top$
Idempotent	$\phi \wedge \phi = \phi$ $\phi \vee \phi = \phi$
Absorption	$\phi_1 \vee (\phi_1 \wedge \phi_2) = \phi_1$ $\phi_1 \wedge (\phi_1 \vee \phi_2) = \phi_1$

Figure 1.4: PC Identities

course simple to prove validity of these identities using truth tables.

Many of these identities are the same as those used in the algebra of real numbers if $\vee$ and $\wedge$ are treated as addition and multiplication respectively; however, differences are the absorption and idempotent laws along with any that have negations. For example, one of the distributivity laws for PC and real numbers directly correspond:

$$\phi_1 \wedge (\phi_2 \vee \phi_3) = (\phi_1 \wedge \phi_2) \vee (\phi_1 \wedge \phi_3)$$
$$x_1 * (x_2 + x_3) = (x_1 * x_2) + (x_1 * x_3)$$

In a manner similar to the algebra over the reals all high school students are familiar with, the identities can also be considered as algebra rules that can be used to prove that $\phi_1 = \phi_2$, or equivalently that $\phi_1 \leftrightarrow \phi_2$ is valid. A useful special case occurs when $\phi_2 \equiv \mathsf{T}$ since this then amounts to proving the validity of ϕ_1. Such an algebraic system was developed by Boole in the 19th century and is thus referred to as Boolean algebra.

Example 1.9: Suppose we wish to prove validity of the wff $\phi \equiv (p \wedge (p \rightarrow q)) \rightarrow q$ using Boolean algebra instead of truth tables as done in Example 1.8. Preprocessing ϕ to remove the implication results in $\neg(p \wedge (\neg p \vee q)) \vee q$. The identities may now be applied:

$$
\begin{aligned}
\neg(p \wedge (\neg p \vee q)) \vee q &= (\neg p \vee \neg(\neg p \vee q)) \vee q && \text{(Duality)} \\
&= (\neg p \vee (\neg\neg p \wedge \neg q)) \vee q && \text{(Duality)} \\
&= (\neg p \vee (p \wedge \neg q)) \vee q && \text{(Double Negation)} \\
&= ((\neg p \vee p) \wedge (\neg p \vee \neg q)) \vee q && \text{(Distributivity)} \\
&= ((p \vee \neg p) \wedge (\neg p \vee \neg q)) \vee q && \text{(Commutativity)} \\
&= (\mathsf{T} \wedge (\neg p \vee \neg q)) \vee q && \text{(Negation)} \\
&= ((\neg p \vee \neg q) \wedge \mathsf{T}) \vee q && \text{(Commutativity)} \\
&= (\neg p \vee \neg q) \vee q && \text{(Identitity)} \\
&= \neg p \vee (\neg q \vee q) && \text{(Associativity)} \\
&= \neg p \vee (q \vee \neg q) && \text{(Commutativity)} \\
&= \neg p \vee \mathsf{T} && \text{(Negation)} \\
&= \mathsf{T} && \text{(Annulment)}
\end{aligned}
$$

Note that the ϕ's in the identities represented wffs (not necessarily atoms), thus allowing us to treat the identities as schemas in the above derivation.

Reducing PC Operator Types

Although our PC syntax had a rich syntax, we earlier showed that the $\rightarrow$ and $\leftrightarrow$ operators could be eliminated. Constants can also be eliminated using the negation identities (among others). Similarly, the duality laws can be used to eliminate either $\wedge$ or $\vee$ This results in the following theorem:

Theorem 1.2 *All PC wffs can be represented using only the operators* $\{\neg, \wedge\}$. *Similarly,* $\{\neg, \vee\}$ *also provide a sufficiently powerful set of operators.*

This raises the question of whether it is possible to reduce to only one operator. This is not possible using the operators we have defined so far, but is possible if there were a $\uparrow$ (not and, also called NAND) or $\downarrow$ (not or, also called NOR) operator:

Theorem 1.3 *All PC wffs can be represented using only* $\{\uparrow\}$*, where* $\phi \uparrow \psi$ *is defined to be* $\neg(\phi \wedge \psi)$*. Similarly, the set* $\{\downarrow\}$ *also suffices, where* $\phi \downarrow \psi$ *is defined to be* $\neg(\phi \vee \psi)$*.*

Proof: We have already shown that $\{\neg, \wedge\}$ is sufficient to express all PC wffs. But $\neg\phi = \phi \uparrow \phi$ and $\phi \wedge \psi = \neg\neg(\phi \wedge \psi) = \neg(\phi \uparrow \psi) = (\phi \uparrow \psi) \uparrow (\phi \uparrow \psi)$. This shows that all PC wffs can be represented using only the $\uparrow$ operator. The $\{\downarrow\}$ case is left as an exercise for the reader.
QED

These theorems effectively state that all PC wffs can be normalized into a form using only a small set of operators.

There are several advantages of normal forms:

- Any proofs of meta-theorems about the logic in the rest of this chapter need only consider a few cases. Since such proofs generally require one case per operator, it is simpler to treat the other operators as abbreviations using the selected primitive operators.

- Proof procedures only need to consider a few operators, for similar reasons to above. Since it is desirable to automate these proof procedures, it may be simpler to construct algorithms/programs that only consider a few cases.

These are balanced by a major disadvantage: Normal forms often lead to an explosion in formula size. Indeed, it is not rare to see the size of formulas have an exponentially larger number of operators than in the original form.

We now turn our attention to a few normal forms that are particularly useful. For all of these, we assume that non-nullary operators other than $\neg$, $\wedge$, and $\vee$ have been eliminated. We first define a **literal** to be either an atom or a negation of an atom, and the literal is said to be either **positive** or **negative** depending on whether it is an atom or the negation of an atom. Examples of literals are p and $\neg p$, but not $\neg(p \vee q)$.

Negation Normal Form (NNF)

A wff is in NNF if any instances of the $\neg$ operator apply only to atoms – *i.e.*, all negations are 'pushed in' as far as possible. To convert a wff into NNF, first observe that since a wff can only have $\wedge$ and $\vee$ operators (assuming $\rightarrow$ and $\leftrightarrow$ have been removed), any $\neg$ can only apply to an atom, constant, $\neg$-wff, $\wedge$-wff, or $\vee$-wff. In the $\neg$ case, the double negation rule is applied to eliminate both negations. In the $\wedge$ and $\vee$ cases, the duality rules can be applied to push the negations in 1 level. Repeatedly applying the above transformations results in an NNF wff.

Example 1.10: Consider the wff $\phi \equiv \neg((p_1 \wedge p_2) \vee \neg p_3)$. Then, ϕ's NNF form can be determined as follows:

$$\begin{aligned}\neg((p_1 \wedge p_2) \vee \neg p_3) &= (\neg(p_1 \wedge p_2) \wedge \neg\neg p_3) \\ &= (\neg(p_1 \wedge p_2) \wedge p_3) \\ &= ((\neg p_1 \vee \neg p_2) \wedge p_3)\end{aligned}$$

Disjunctive Normal Form (DNF)

A wff is said to be in DNF form if it is a disjunction of conjunctions of literals – that is, one of the form:

$$(\ell_{1,1} \wedge \ell_{1,2} \wedge \ldots \wedge \ell_{1,n_1}) \vee (\ell_{2,1} \wedge \ell_{2,2} \wedge \ldots \wedge \ell_{2,n_2}) \vee \ldots \vee (\ell_{m,1} \wedge \ell_{m,2} \wedge \ldots \wedge \ell_{m,n_m})$$

where each $\ell_{i,j}$ is a literal. Each of the disjuncts is called a **clause**. Clearly, every DNF formula is also in NNF. Following usual convention, the given form is strictly speaking not well formed since it leaves out parentheses inside each clause – this does not introduce semantic ambiguity since $\wedge$ is associative. It is simple to see that every wff ϕ has a DNF form since we can draw a truth table for ϕ where each disjunct corresponds to one model (truth table row) satisfying ϕ.

Example 1.11: Consider the wff ϕ given by the following truth table:

p_1	p_2	p_3	ϕ
$\bot$	$\bot$	$\bot$	$\bot$
$\bot$	$\bot$	$\top$	$\top$
$\bot$	$\top$	$\bot$	$\bot$
$\bot$	$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$	$\top$
$\top$	$\bot$	$\top$	$\top$
$\top$	$\top$	$\bot$	$\top$
$\top$	$\top$	$\top$	$\top$

Then, the DNF representation can be read off each model/row of the table evaluating to $\top$, resulting in: $(\neg p_1 \wedge \neg p_2 \wedge p_3) \vee (\neg p_1 \wedge p_2 \wedge p_3) \vee (p_1 \wedge \neg p_2 \wedge \neg p_3) \vee (p_1 \wedge \neg p_2 \wedge p_3) \vee (p_1 \wedge p_2 \wedge \neg p_3) \vee (p_1 \wedge p_2 \wedge p_3)$

Conjunctive Normal Form (CNF)

The CNF form is similar to DNF, but requires that the wff is a conjunction of disjunction of literals – that is, one of the form:

$$(\ell_{1,1} \vee \ell_{1,2} \vee \ldots \vee \ell_{1,n_1}) \wedge (\ell_{2,1} \vee \ell_{2,2} \vee \ldots \vee \ell_{2,n_2}) \wedge \ldots \wedge (\ell_{m,1} \vee \ell_{m,2} \vee \ldots \vee \ell_{m,n_m})$$

where each $\ell_{i,j}$ is a literal. Here, each conjunct is called a **clause**. As with DNF, every CNF formula is also in NNF, and identities can be used for the transformation. The truth table approach can also be used, though the justification is not as obvious as with DNF. In DNF, the truth table approach readily followed since each row is a model and there is an implicit disjunction between the rows. There are two key observations to generate the CNF form of a truth table for ϕ:

1. Consider a row in ϕ's truth table, and let it correspond to $\ell_1 \wedge ... \wedge \ell_k$ where the ℓ_i's are literals. The negation of this is $\neg\ell_1 \vee ... \vee \neg\ell_k$, which has the form of a CNF clause and also falsifies the model. Thus, we can construct a CNF clause that falsifies any particular model by simply selecting those atoms that are $\bot$.

2. A CNF wff is false iff at least one of its clauses is false.

Thus, we need only consider all the non-model rows of the truth table and ensure that each of these has a clause falsifying it.

Example 1.12: Continuing with the truth table of Example 1.11, there are two non-model rows. Thus, a CNF wff is $(p_1 \vee p_2 \vee p_3) \wedge (p_1 \vee \neg p_2 \vee p_3)$.

Although the truth table approach provides a simple and automatable method to convert wffs to DNF/CNF, these wffs are not necessarily optimal – indeed, Examples 1.11 and 1.12 could also have been represented by the much simpler wff $p_1 \vee p_3$, which is in both DNF and CNF form (with 2 and 1 clauses respectively). Since there are an exponential number of models, converting a wff to a normal form may result in exponentially greater size. In the example, the shorter form was also in DNF/CNF, but this is not true in general. The problem of minimizing DNF/CNF wffs is particularly important in hardware design since most hardware units correspond to DNF/CNF wffs, and their size depends on the number of operators. There are thus several algorithms for this minimization, though this book does not cover these since it is not our focus.

Avoiding Truth Tables

Real problems often have many atoms, making it impractical to construct a complete truth table before conversion to a normal form. It is thus desirable to generate algorithms that can do the conversions. We first formalize the

elimination of the $\rightarrow$ and $\leftrightarrow$ operators using a recursive algorithm (called PRE, standing for preprocess) that enumerates all forms of wffs[8]:

function PRE(ϕ)
// Precondition: ϕ is an arbitrary PC wff
// Postcondition: The returned wff is equivalent to ϕ
// but contains only $\neg$, $\wedge$, and $\vee$ operators
  if ϕ is a constant or atom **then return** ϕ
  else if $\phi = \neg\psi_1$ **then return** $\neg$ PRE(ψ_1)
  else if $\phi = \psi_1 \wedge \psi_2$ **then return** PRE(ψ_1) $\wedge$ PRE(ψ_2)
  else if $\phi = \phi_1 \vee \phi_2$ **then return** PRE(ϕ_1) $\vee$ PRE(ϕ_2)
  else if $\phi = \phi_1 \rightarrow \phi_2$ **then return** $\neg$PRE(ϕ_1) $\vee$ PRE(ϕ_2)
  else if $\phi = \phi_1 \leftrightarrow \phi_2$ **then return**
    ($\neg$PRE(ϕ_1) $\vee$ PRE(ϕ_2)) $\wedge$ ($\neg$PRE(ϕ_2) $\vee$ PRE(ϕ_1))
  end if
end function

Throughout this book, we use the convention of explicitly stating function pre- and post-conditions to document function semantics. More precisely, if the precondition holds before calling the function, the postcondition holds after execution of the function. When coding such algorithms, it is also important to ensure that all cases in the syntax are covered, though its correctness (*i.e.*, compliance with pre-/post-conditions) is otherwise simple to prove.

We have already seen an NNF conversion procedure that is readily automatable as long as input has been preprocessed. It can be algorithmically expressed as:

function NNF(ϕ)
// Precondition: ϕ is a wff containing no $\rightarrow$ or $\leftrightarrow$ operators
// Postcondition: The NNF form of ϕ is returned
  if ϕ is a constant or literal **then return** ϕ
  else if $\phi = \phi_1 \vee \phi_2$ **then return** NNF(ϕ_1) $\vee$ NNF(ϕ_2)
  else if $\phi = \phi_1 \wedge \phi_2$ **then return** NNF(ϕ_1) $\wedge$ NNF(ϕ_2)
  else if $\phi = \neg\neg\phi_1$ **then return** NNF(ϕ_1)
  else if $\phi = \neg(\phi_1 \vee \phi_2)$ **then return** NNF($\neg\phi_1$) $\wedge$ NNF($\neg\phi_2$)
  else if $\phi = \neg(\phi_1 \wedge \phi_2)$ **then return** NNF($\neg\phi_1$) $\vee$ NNF($\neg\phi_2$)
  end if
end function

For converting to CNF, we assume that the wff has been converted to NNF, and observe that the conjunction of two CNF wffs is in CNF. This leaves the case where we have the disjunction of two CNF wffs. The distributivity identity

[8] In this text we fully state all cases for clarity, even though the last case need not be explicitly stated, as it only applies if all other cases fail.

can be applied to reduce this case – for example:

$$\begin{aligned}&(p_1 \wedge (p_2 \vee p_3)) \vee (q_1 \wedge q_2)\\ &= ((p_1 \wedge (p_2 \vee p_3)) \vee q_1) \wedge ((p_1 \wedge (p_2 \vee p_3)) \vee q_2)\\ &= (p_1 \vee q_1) \wedge (p_2 \vee p_3 \vee q_1) \wedge (p_1 \vee q_2) \wedge (p_2 \vee p_3 \vee q_2)\end{aligned}$$

which is in CNF. This results in the following algorithms:

function CNF(ϕ)
// Precondition: ϕ has been preprocessed and is also in NNF
// Postcondition: The CNF form of ϕ is returned
  if $\phi = \phi_1 \wedge \phi_2$ **then return** CNF(ϕ_1) $\wedge$ CNF(ϕ_2)
  else if $\phi = \phi_1 \vee \phi_2$ **then return** DIST(CNF(ϕ_1),CNF(ϕ_2))
  else return ϕ
  end if
end function
function DIST(ϕ_1,ϕ_2)
// Precondition: ϕ_1 and ϕ_2 are in CNF
// Postcondition: The CNF form of $\phi_1 \vee \phi_2$ is returned
  if $\phi_1 = \phi_{11} \wedge \phi_{12}$ **then return** DIST(ϕ_{11},ϕ_2) $\wedge$ DIST(ϕ_{12}, ϕ_2)
  else if $\phi_2 = \phi_{21} \wedge \phi_{22}$ **then return** DIST(ϕ_1,ϕ_{21}) $\wedge$ DIST(ϕ_1,ϕ_{22})
  else return $\phi_1 \vee \phi_2$
  end if
end function

Note that we do not need a negation case since ϕ is already in NNF. Also note that wffs such as $\phi_{11} \wedge \phi_{12}$ only restrict to *at least* two conjuncts, as ϕ_{11} and ϕ_{12} themselves might be conjunctions. Our algorithm has not removed the constants $\bot$ and $\top$, though it is simple enough to post-process the resulting CNF wff using the identity and annulment rules to remove constants[9]. It also may result in clauses that contain both an atom and its negation, which can obviously be further simplified. Again, post-processing can be used for this.

A call to CNF(NNF(PRE(ϕ))) can be used to convert any wff to CNF. If the goal is conversion to DNF, a similar algorithm can be used since the other form of distributivity applies to the DNF analog. In any case, the existence of these terminating algorithms yields the following theorem:

Theorem 1.4 *Any PC wff is equivalent to some CNF wff. The same also holds for DNF.*

Example 1.13: We wish to convert to CNF

$$\phi \equiv (p_1 \rightarrow p_2) \vee (\neg(p_3 \rightarrow p_4) \wedge p_5)$$

[9]Strictly speaking, PRE should remove constants to be consistent with the CNF definition. However, we have excluded this part for readability purposes since it would add cases for each possible way a constant can appear. Alternatively, $\top$ may first be replaced by an equivalent wff such as $p \vee \neg p$ and similarly for $\bot$, and the given algorithm may then be applied.

using our algorithm. We first preprocess to eliminate $\rightarrow$:

$$
\begin{aligned}
&PRE((p_1 \rightarrow p_2) \vee (\neg(p_3 \rightarrow p_4) \wedge p_5)) \\
&= PRE(p_1 \rightarrow p_2) \vee PRE(\neg(p_3 \rightarrow p_4) \wedge p_5) \\
&= (\neg\, PRE(p_1) \vee PRE(p_2)) \vee PRE(\neg(p_3 \rightarrow p_4) \wedge p_5) \\
&= (\neg\, PRE(p_1) \vee PRE(p_2)) \vee (PRE(\neg(p_3 \rightarrow p_4)) \wedge PRE(p_5)) \\
&= (\neg\, PRE(p_1) \vee PRE(p_2)) \vee (\neg\, PRE(p_3 \rightarrow p_4) \wedge PRE(p_5)) \\
&= (\neg\, PRE(p_1) \vee PRE(p_2)) \vee ((\neg(\neg\, PRE(p_3) \vee PRE(p_4))) \wedge PRE(p_5)) \\
&= \ldots \\
&= (\neg\, p_1 \vee p_2) \vee (\neg(\neg\, p_3 \vee p_4) \wedge p_5)
\end{aligned}
$$

Note that in each step of the algorithm, we identified the primary operator of the wff and mechanistically applied the appropriate case of the algorithm. The appropriate connective can also be identified by scanning the wff's formation tree representation. Next, we convert to NNF:

$$
\begin{aligned}
&NNF((\neg\, p_1 \vee p_2) \vee (\neg(\neg\, p_3 \vee p_4) \wedge p_5)) \\
&= NNF(\neg\, p_1 \vee p_2) \vee NNF(\neg(\neg\, p_3 \vee p_4) \wedge p_5) \\
&= (NNF(\neg\, p_1) \vee NNF(p_2)) \vee NNF(\neg(\neg\, p_3 \vee p_4) \wedge p_5) \\
&= (NNF(\neg\, p_1) \vee NNF(p_2)) \vee (NNF(\neg(\neg\, p_3 \vee p_4)) \wedge NNF(p_5)) \\
&= (NNF(\neg\, p_1) \vee NNF(p_2)) \vee ((NNF(\neg\neg\, p_3) \wedge NNF(\neg\, p_4)) \wedge NNF(p_5)) \\
&= (NNF(\neg\, p_1) \vee NNF(p_2)) \vee ((NNF(p_3) \wedge NNF(\neg\, p_4)) \wedge NNF(p_5)) \\
&= \ldots \\
&= (\neg\, p_1 \vee p_2) \vee ((p_3 \wedge \neg\, p_4) \wedge p_5)
\end{aligned}
$$

Now, we are ready to call CNF:

$$
\begin{aligned}
&CNF((\neg\, p_1 \vee p_2) \vee ((p_3 \wedge \neg\, p_4) \wedge p_5)) \\
&= DIST(CNF(\neg\, p_1 \vee p_2), CNF((p_3 \wedge \neg\, p_4) \wedge p_5)) \\
&= DIST(CNF(\neg\, p_1 \vee p_2), (CNF(p_3 \wedge \neg\, p_4) \wedge CNF(p_5))) \\
&= DIST(CNF(\neg\, p_1 \vee p_2), (CNF(p_3) \wedge CNF(\neg\, p_4)) \wedge CNF(p_5)) \\
&= DIST(DIST(CNF(\neg\, p_1), CNF(p_2)), (CNF(p_3) \wedge CNF(\neg\, p_4)) \wedge CNF(p_5)) \\
&= \ldots \\
&= DIST(DIST(\neg\, p_1, p_2), ((p_3 \wedge \neg\, p_4) \wedge p_5)) \\
&= DIST(\neg\, p_1 \vee p_2, ((p_3 \wedge \neg\, p_4) \wedge p_5)) \\
&= DIST(\neg\, p_1 \vee p_2, p_3 \wedge \neg\, p_4) \wedge DIST(\neg\, p_1 \vee p_2, p_5) \\
&= DIST(\neg\, p_1 \vee p_2, p_3 \wedge \neg\, p_4) \wedge (\neg\, p_1 \vee p_2 \vee p_5) \\
&= (DIST(\neg\, p_1 \vee p_2, p_3) \wedge DIST(\neg\, p_1 \vee p_2, \neg\, p_4)) \wedge (\neg\, p_1 \vee p_2 \vee p_5) \\
&= \ldots \\
&= (\neg\, p_1 \vee p_2 \vee p_3) \wedge (\neg\, p_1 \vee p_2 \vee \neg\, p_4) \wedge (\neg\, p_1 \vee p_2 \vee p_5)
\end{aligned}
$$

In our derivation, we have been lax in allowing conjunctions and disjunctions of more than 2 operands (which are not well formed, technically speaking) for readability reasons. However, we have shown the gory details of every step to emulate how a program might execute the algorithm, even though paper derivations normally skip most steps.

Now consider a slight modification of the above example replacing p_5 with $(p_5 \vee \ldots \vee p_{99})$:

$$\phi \equiv (p_1 \rightarrow p_2) \vee (\neg(p_3 \rightarrow p_4) \wedge (p_5 \vee \ldots \vee p_{99}))$$

Then, a truth table would require 2^{99} rows, which is clearly not feasible, while the algorithm would quickly produce

$$(\neg\ p_1 \vee p_2 \vee p_3) \wedge (\neg\ p_1 \vee p_2 \vee \neg\ p_4) \wedge (\neg\ p_1 \vee p_2 \vee p_5 \vee \ldots \vee p_{99})$$

Horn Formulas

The rest of this chapter will discuss both the high complexity of the PC SAT and VAL problems along with techniques that work well in many cases. However, some PC wffs are representable in restricted forms that afford a much lower complexity. One such form is a restriction of CNF named after the logician Alfred Horn, or Horn Formulas (HFs).

Def: A **Horn Clause** is a clause that has at most one positive literal (and any number of negative literals). A CNF formula is a **Horn Formula** if each clause is a Horn Clause.
There are thus three types of Horn clauses:

1. Those with one positive literal and no negative literals, *e.g.*, *rain* and *outside* (to signify that it is raining and I am outside, respectively)
2. Those with one positive literal and at least one negative literal, *e.g.*, $getwet \vee \neg rain \vee \neg outside$
3. Those with no positive literals, *e.g.*, $\neg getwet$

All wffs are not representable in Horn form, but there are many applications where HFs are useful. Section 1.11.1 shows how the above distinction into three forms can be exploited for an efficient and automatable inference mechanism, and Section 3.9 later shows how the restriction serves as the basis of a class of programming languages.

1.8 Game Theoretic Semantics of PC

This section will give a game theoretic semantics of PC, assuming that the $\rightarrow$ and $\leftrightarrow$ operators have been removed using the techniques of Section 1.7. Recall that in game theoretic semantics, the meaning of a wff is given as a result of

a game between a truthifier and falsifier. There are two players, $\boldsymbol{M}$ and $\boldsymbol{N}$ standing for myself and nature respectively, in keeping with Hintikka's original convention. Initially, $\boldsymbol{M}$ is the truthifier and $\boldsymbol{N}$ is the falsifier. The model $\mathcal{M}$ is given. In each round of the game, the truthifier at that point of the game attempts to show that the wff at that point is true in $\mathcal{M}$, while the falsifier attempts the opposite.

Stated more formally, we wish to generate the semantics of a wff ϕ in model $\mathcal{M}$, using a two-player game $\mathcal{G}_{\mathcal{M}}(\phi)$. Formally, there are three rules of this game, and the rule corresponding to the primary operator of the wff is chosen in each round:

$\mathrm{R}_{\vee}$ ($\mathcal{G}_{\mathcal{M}}(\phi_1 \vee \phi_2)$): The truthifier chooses one of the two disjuncts ϕ_i ($i = 1$ or 2), and the game reduces to the new game $\mathcal{G}_{\mathcal{M}}(\phi_i)$.

$\mathrm{R}_{\wedge}$ ($\mathcal{G}_{\mathcal{M}}(\phi_1 \wedge \phi_2)$): The falsifier chooses one of the two conjuncts ϕ_i ($i = 1$ or 2), and the game reduces to the new game $\mathcal{G}_{\mathcal{M}}(\phi_i)$.

$\mathrm{R}_{\neg}$ ($\mathcal{G}_{\mathcal{M}}(\neg\phi)$): The game reduces to the new game $\mathcal{G}_{\mathcal{M}}(\phi)$, but the roles of the players $\boldsymbol{M}$ and $\boldsymbol{N}$ flip. This flipping of players means that if $\boldsymbol{M}$ is the current truthifier, it becomes the falsifier and $\boldsymbol{N}$ becomes the truthifier (and vice versa).

The game ends when the wff is atomic – *i.e.*, at the game $\mathcal{G}_{\mathcal{M}}(a)$, where $a \in \mathcal{A}$. The truthifier at the end of the game wins iff a is true in $\mathcal{M}$, and the status is reversed for the falsifier.

The game is guaranteed to terminate since each of the above rules reduces to a game whose argument wff is a [strict] subformula of the original wff and the original wff is finite. A player $\boldsymbol{P\ell}$ is said to have a winning strategy if $\boldsymbol{P\ell}$ can make choices to ensure victory regardless of what choices the other player makes. If player $\boldsymbol{M}$ has a winning strategy for $\mathcal{G}_{\mathcal{M}}(\phi)$, ϕ is concluded to be true in $\mathcal{M}$. Otherwise, $\boldsymbol{N}$ has a winning strategy and ϕ is concluded to be false in $\mathcal{M}$.

Example 1.14: Consider the wff $\phi \equiv (p_1 \wedge \neg p_2) \vee \neg p_1$ and the model $\mathcal{M}$ with $\mathcal{L}_{\mathcal{M}} = \{p_1\}$. Initially $\boldsymbol{M}$ is the truthifier and $\boldsymbol{N}$ is the falsifier. In the first round, the disjunction rule dictates that the truthifier $\boldsymbol{M}$ picks one of the disjuncts, say $p_1 \wedge \neg p_2$. Continuing, one game play of $\mathcal{G}_{\mathcal{M}}((p_1 \wedge \neg p_2) \vee \neg p_1)$ is:

Round	**Wff**	**Truthifier**	**Falsifier**	Comments
0	$(p_1 \wedge \neg p_2) \vee \neg p_1$	$\boldsymbol{M}$	$\boldsymbol{N}$	Initial
1	$(p_1 \wedge \neg p_2)$	$\boldsymbol{M}$	$\boldsymbol{N}$	$R_\vee$ (by $\boldsymbol{M}$)
2	$\neg p_2$	$\boldsymbol{M}$	$\boldsymbol{N}$	$R_\wedge$ (by $\boldsymbol{N}$)
3	p_2	$\boldsymbol{N}$	$\boldsymbol{M}$	$R_\neg$

The falsifier $\boldsymbol{M}$ thus wins this game since p_2 is false in $\mathcal{M}$. However, it is possible that $\boldsymbol{N}$ had made a different choice in round 2, resulting in the following game:

Round	**Wff**	**Truthifier**	**Falsifier**	Comments
0	$(p_1 \wedge \neg p_2) \vee \neg p_1$	$\boldsymbol{M}$	$\boldsymbol{N}$	Initial
1	$(p_1 \wedge \neg p_2)$	$\boldsymbol{M}$	$\boldsymbol{N}$	$R_\vee$ (by $\boldsymbol{M}$)
2	p_1	$\boldsymbol{M}$	$\boldsymbol{N}$	$R_\wedge$ (by $\boldsymbol{N}$)

This game also results in a win for $\boldsymbol{M}$. Since we have covered all choices that $\boldsymbol{N}$ can take, $\boldsymbol{M}$ has a strategy to ensure victory and it can be concluded that $\mathcal{M} \models (p_1 \wedge \neg p_2) \vee \neg p_1$.

Note that each game in the above example corresponds to only one possible strategy. For example, had $\boldsymbol{M}$ used a strategy that chose the $\neg p_1$ disjunct in the first round, $\boldsymbol{N}$ would have won. However, only one winning strategy is needed to determine that $\mathcal{M} \models \phi$.

It is not generally possible to determine a winning strategy even if one exists. A single game play is determined by the choices made by the players and can not show anything about the existence of a winning strategy. In the PC case, there are fortunately only finitely many choices and a search through all resulting game plays can be used to check if one of them corresponds to a winning strategy.

The set of all game plays can also be drawn as a game tree that captures all possible game plays, with each path down the tree corresponding to one possible game. For example, Figure 1.5 illustrates the game tree to show that the model assigning $\bot$ to p_1 and $\top$ to p_2 satisfies the wff $\neg((p_1 \wedge \neg p_2) \wedge (\neg p_1 \vee \neg p_2)) \wedge p_2$. Oval nodes are nodes in which $\boldsymbol{N}$ makes a choice, while rectangular nodes involve either no choice or a choice by $\boldsymbol{M}$. Nodes in this game tree are labeled with the truthifier to the left and falsifier to the right for convenience, and a few edges (1A,1B,2A,2B) are labeled for reference. The wff is satisfied if $\boldsymbol{M}$ has a winning strategy regardless of choices made by $\boldsymbol{N}$, or in other words, it is possible to force a win by $\boldsymbol{M}$ for every branch from an oval node. In this example, the first choice is between the edges labeled by 1A and 1B.

- If $\boldsymbol{N}$ chooses 1B, $\boldsymbol{M}$ wins immediately.
- In the 1A case, $\boldsymbol{M}$ is later faced with a choice between 2A and 2B. If $\boldsymbol{M}$ chooses 2B, it is possible for $\boldsymbol{N}$ to make choices ensuring a win; however $\boldsymbol{M}$ can choose 2A to avoid this and ensure victory.

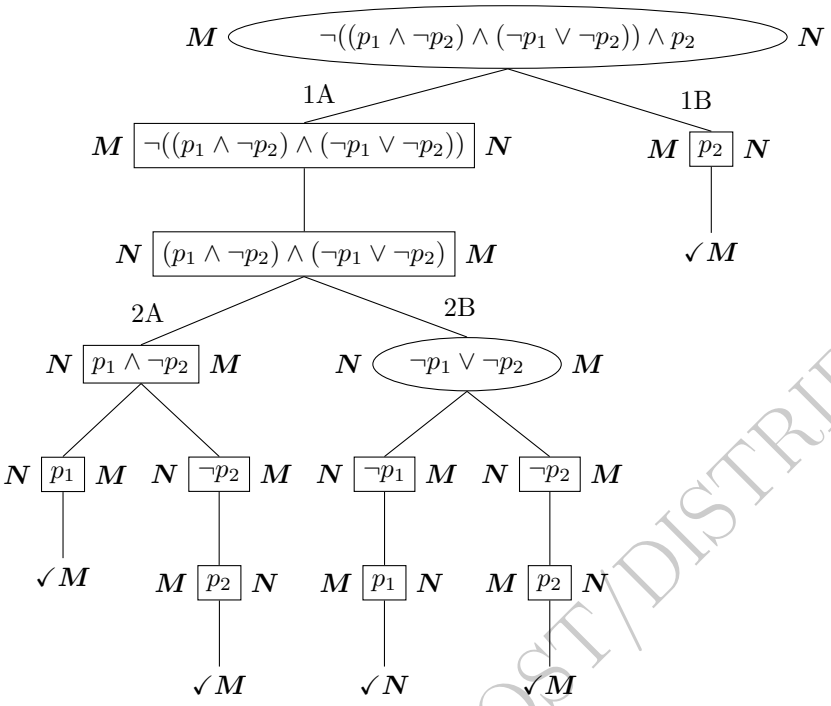


Figure 1.5: Game Tree for $\neg((p_1 \wedge \neg p_2) \wedge (\neg p_1 \vee \neg p_2)) \wedge p_2$ in Model $\{p_2\}$

Thus, $\boldsymbol{M}$ has a strategy to win the game, which is consistent with the model theoretic semantics.

While game theoretic semantics may seem to be significantly more complex than the model theoretic approach, they have some benefits. Some may see it as more natural to use game rules instead of recursive reduction as in the model theoretic approach. Some extensions of PC may also be more naturally explained as game rules – for example, the semantics of a PC extension in which a particular subformula may only be used a fixed number of times, might be more naturally expressed as a game rule. The central role of strategies introduces new ways of conceptualizing logics and explaining *why* certain wffs are satisfied. There are also applications best modeled as games between a protagonist and a neutral environment or antagonistic rogue, with the latter particularly appropriate in modeling security. Some non-classical logics may also be more amenable to game theoretic semantics.

1.9 Sequent Calculus

Recall that a proof theoretic semantics of a logic corresponds to a mechanism for showing validity of wffs in the logic. Traditionally, this mechanism took the

form of a set of axiom schemas and inference rules that can be used to prove validity of wffs. Whereas axiom schemas may be instantiated to assert a base set of valid wffs, inference rules are used to infer valid wffs from other wffs. The sequent calculus was originally developed in 1935 by the logician Gentzen as one such proof theoretic semantics, and several variants have since been developed, typically for first order logic. This section presents one version of it named PK, which is a specialization of the first order logic sequent calculus discussed in Section 3.12 and called LK.

A **sequent** is an expression

$$\Gamma \vdash_{SC} \Delta$$

where Γ and Δ are finite sequences of wffs. Γ and Δ may be empty as long as both are not empty. The symbol $\vdash_{SC}$ is read [single] "turnstile" to distinguish from the double turnstile of model theoretic semantics. Informally, the sequent

$$\phi_1, \phi_2, \ldots, \phi_m \vdash_{SC} \psi_1, \psi_2, \ldots, \psi_n$$

means that the disjunction of the ψ_i's can be derived/proven from the conjunction of the ϕ_i's, which can be shown equivalent to:

$$\vdash_{SC} \bigwedge_{i=1}^{m} \phi_i \rightarrow \bigvee_{i=1}^{n} \psi_i$$

i.e., the wff on the right side of the turnstile can be derived from an empty wff (see Section 1.13 for the Deduction Theorem which shows the above equivalence). Logical identities from earlier can be applied to show this also equivalent to:

$$\vdash_{SC} (\neg\phi_1 \vee \neg\phi_2 \vee \ldots \vee \neg\phi_m \vee \psi_1 \vee \psi_2 \vee \ldots \vee \psi_n)$$

This latter form also explains the seeming oddity of sequent semantics where commas on the left side of the turnstile represent conjunction while those on the right side represent disjunction. However, the reader should not confuse $\vdash_{SC}$ with $\rightarrow$ as they refer to provability and implication respectively, and the two are not necessarily equivalent in all logics (Section 1.13 will discuss this further).

In the sequent calculus presented here, we assume for simplicity that the symbols $\top$, $\bot$, and $\leftrightarrow$ have been eliminated, though there are variations that directly support these. This may be done using the identities of Section 1.7, effectively treating these symbols as abbreviations and not proper syntactic entities of PC. Notationally, upper case Greek letters are used to indicate sequences of wffs (lower case Greek letters still represent individual wffs as before). Recall that sequent calculus consists of axioms and rules. In PK, there is just one axiom schema, indicating that any wff follows from itself:

$$\frac{}{\phi \vdash_{SC} \phi} \text{ (Ax)}$$

Figure 1.6 shows the rules of the sequent calculus.

	Left	Right
Structural Rules		
Weakening	$\dfrac{\Gamma \vdash_{SC} \Delta}{\phi, \Gamma \vdash_{SC} \Delta}$ (Wk L)	$\dfrac{\Gamma \vdash_{SC} \Delta}{\Gamma \vdash_{SC} \Delta, \phi}$ (Wk R)
Contraction	$\dfrac{\phi, \phi, \Gamma \vdash_{SC} \Delta}{\phi, \Gamma \vdash_{SC} \Delta}$ (Con L)	$\dfrac{\Gamma \vdash_{SC} \Delta, \phi, \phi}{\Gamma \vdash_{SC} \Delta, \phi}$ (Con R)
Exchange	$\dfrac{\Pi, \phi, \psi, \Gamma \vdash_{SC} \Delta}{\Pi, \psi, \phi, \Gamma \vdash_{SC} \Delta}$ (Exc L)	$\dfrac{\Gamma \vdash_{SC} \Sigma, \phi, \psi, \Delta}{\Gamma \vdash_{SC} \Sigma, \psi, \phi, \Delta}$ (Exc R)
Cut	$\dfrac{\Gamma \vdash_{SC} \Delta, \phi \quad \phi, \Sigma \vdash_{SC} \Pi}{\Gamma, \Sigma \vdash_{SC} \Delta, \Pi}$ (Cut)	
Logical Rules		
Negation	$\dfrac{\Gamma \vdash_{SC} \Delta, \phi}{\neg\phi, \Gamma \vdash_{SC} \Delta}$ ($\neg$L)	$\dfrac{\phi, \Gamma \vdash_{SC} \Delta}{\Gamma \vdash_{SC} \Delta, \neg\phi}$ ($\neg$R)
Conjunction	$\dfrac{\phi, \Gamma \vdash_{SC} \Delta}{\phi \wedge \psi, \Gamma \vdash_{SC} \Delta}$ ($\wedge$L) $\quad \dfrac{\psi, \Gamma \vdash_{SC} \Delta}{\phi \wedge \psi, \Gamma \vdash_{SC} \Delta}$ ($\wedge$L)	$\dfrac{\Gamma \vdash_{SC} \Delta, \phi \quad \Gamma \vdash_{SC} \Delta, \psi}{\Gamma \vdash_{SC} \Delta, \phi \wedge \psi}$ ($\wedge$R)
Disjunction	$\dfrac{\phi, \Gamma \vdash_{SC} \Delta \quad \psi, \Gamma \vdash_{SC} \Delta}{\phi \vee \psi, \Gamma \vdash_{SC} \Delta}$ ($\vee$L)	$\dfrac{\Gamma \vdash_{SC} \Delta, \phi}{\Gamma \vdash_{SC} \Delta, \phi \vee \psi}$ ($\vee$R) $\quad \dfrac{\Gamma \vdash_{SC} \Delta, \psi}{\Gamma \vdash_{SC} \Delta, \phi \vee \psi}$ ($\vee$R)
Implication	$\dfrac{\psi, \Gamma \vdash_{SC} \Delta \quad \Gamma \vdash_{SC} \Delta, \phi}{\phi \to \psi, \Gamma \vdash_{SC} \Delta}$ ($\to$ L)	$\dfrac{\phi, \Gamma \vdash_{SC} \Delta, \psi}{\Gamma \vdash_{SC} \Delta, \phi \to \psi}$ ($\to$ R)

Figure 1.6: Sequent Calculus Rules for PC

Rules are applied to sequent/s by instantiating the sequent/s to the top of the rule (called rule **premises**) and reducing to the bottom of the rule (called rule **conclusions**). For example, the ¬L rule may be applied as follows:

$$\frac{\vdash_{SC} r_1, r_1 \vee r_3, p \wedge q}{\neg(p \wedge q) \vdash_{SC} r_1, r_1 \vee r_3} \ (\neg\text{L})$$

Here, Γ and Δ have been instantiated to the empty sequence and "$r_1, r_1 \vee r_3$" respectively.

Rules are partitioned into structural and logical rules – whereas structural rules are used to manipulate sequents by adding, removing, and rearranging wffs, logical rules allow for inference of sequents containing more complicated wffs from simpler sequents. Almost all rules have a left and right version,

corresponding to the cases where the inferred sequents are on the left or right side of the sequent, providing a pleasing symmetry.

Before discussing the usage of sequent calculus, we first need to distinguish English proofs (*e.g.*, proofs of theorems about sequent calculus) from proofs of sequents using sequent calculus – to avoid confusion, we will use the word "derivation" (of a sequent) instead of "proof" for the latter. Additionally, this derivation refers to the derivation of one sequent from one or more other sequents, which is slightly different from our earlier use of the word to indicate that the sequence of wffs Δ can be derived from the sequence of wffs Γ in the sequent $\Gamma \vdash_{SC} \Delta$.

Given a sequent $\Gamma \vdash_{SC} \Delta$, the rules of sequent calculus are used to generate a derivation tree starting with only axioms and terminating with $\Gamma \vdash_{SC} \Delta$ at the root of the tree. The ($\wedge$L) and ($\vee$R) rules have two instances, and derivations will typically need to use just one of these instances. Rules with two sequents as premises ($\wedge$R, $\vee$L, $\rightarrow$L) correspond to the cases where the tree branches, and derivations of both branches are needed.

It is important to note that sequents are syntactic entities and may not be manipulated except through the given rules. For example, the two sequents $\Gamma \vdash_{SC} \phi, \psi$ and $\Gamma \vdash_{SC} \psi, \phi$ are not the same, and the exchange rule must explicitly be used to derive one from the other.

Example 1.15: Consider the standard modus ponens argument, one case of which is represented using the sequent

$$\phi \rightarrow \psi, \phi \vdash_{SC} \psi$$

Working backwards from the sequent, it is clear that the ($\rightarrow$L) rule applies here with Γ being ϕ and Δ being ψ. Applying ($\rightarrow$L) reduces the problem of deriving $\phi \rightarrow \psi, \phi \vdash_{SC} \psi$ to deriving the two sequents:

$$\psi, \phi \vdash_{SC} \psi \quad \text{and} \quad \phi \vdash_{SC} \psi, \phi$$

Continuing, this results in the final derivation tree:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\psi \vdash_{SC} \psi}}{\phi, \psi \vdash_{SC} \psi}\,(\text{Wk L})}{\psi, \phi \vdash_{SC} \psi}\,(\text{Exc L}) \qquad \dfrac{\dfrac{\dfrac{}{\phi \vdash_{SC} \phi}}{\phi \vdash_{SC} \phi, \psi}\,(\text{Wk R})}{\phi \vdash_{SC} \psi, \phi}\,(\text{Exc R})}{\phi \rightarrow \psi, \phi \vdash_{SC} \psi}\,(\rightarrow\text{L})$$

Since the top sequents are all instantiations of the single axiom, the derivation is complete.

As seen in the above example, it is advisable to work backwards from the desired sequent when constructing derivations, using structural rules to manipulate in-

termediate sequents into forms for which axioms or logical rules apply.

While the rules provide a method to derive a sequent, it is quite possible to apply the rules in a way that does not yield the desired sequent. Unlike the above example, the choice of rules to apply is not always obvious, and derivations may require some tact (and plenty of practice).

Example 1.16: Consider the law of the excluded middle, which states that every proposition is either true or false: $\vdash_{SC} p \vee \neg p$. The seemingly obvious approach working backwards is to apply ($\vee$R), thus requiring one of the two sequents $\vdash_{SC} p$ or $\vdash_{SC} \neg p$ to be derived. Clearly, neither holds and they will not be derivable (assuming sequent calculus is sound). However, applying contraction first yields the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{p \vdash_{SC} p}}{p \vdash_{SC} p \vee \neg p}\,(\vee\text{R})}{\vdash_{SC} p \vee \neg p, \neg p}\,(\neg\text{R})}{\vdash_{SC} p \vee \neg p, p \vee \neg p}\,(\vee\text{R})}{\vdash_{SC} p \vee \neg p}\,(\text{Con R})$$

The above example shows that some creativity is needed in constructing derivations in the sequent calculus, since contraction was needed before applying the obvious logical rule. This use of contraction is a common pattern when a particular wff is needed multiple times in a derivation tree – in the above example, $p \vee \neg p$ was needed even after it was decomposed once.

The examples so far have not used the cut rule. If Δ is empty, the cut rule is simply:

$$\dfrac{\Gamma \vdash_{SC} \phi \quad \phi, \Sigma \vdash_{SC} \Pi}{\Gamma, \Sigma \vdash_{SC} \Pi}$$

In this version, it is clear that ϕ acts as a lemma in showing that Π follows from the wffs in Γ and Σ. Similar forms can be seen if other elements are null. This use of lemmas often shortens derivations as can be seen in the following example.

Example 1.17: Consider the sequent $p, p \to q, q \to r \vdash_{SC} r$ One possible derivation of this without using cut is below (other possibilities exist):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdots}}{q \to r, q \vdash_{SC} r}\,(\text{Exc L})}{q, q \to r \vdash_{SC} r}\,(\text{Wk L})}{p, q, q \to r \vdash_{SC} r}\,(\text{Exc L})}{q, p, q \to r \vdash_{SC} r}\,(\text{Exc L})}{q, q \to r, p \vdash_{SC} r} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{p \vdash_{SC} p}}{p \vdash_{SC} p, r}\,(\text{Wk R})}{p \vdash_{SC} r, p}\,(\text{Exc R})}{q \to r, p \vdash_{SC} r, p}\,(\text{Wk L})}{p \to q, q \to r, p \vdash_{SC} r}\,(\to\text{L})}{p \to q, p, q \to r \vdash_{SC} r}\,(\text{Exc L})}{p, p \to q, q \to r \vdash_{SC} r}\,(\text{Exc L})$$

The incomplete part of the above derivation is similar to that of Example 1.15.

Observing that q follows from p, $p \to q$, and q is also sufficient together with $q \to r$ to infer r, the wff q seems to be a good candidate as an intermediate lemma applying cut as follows (with empty Δ):

$$\underbrace{p, p \to q}_{\Gamma}, \underbrace{q \to r}_{\Sigma} \vdash_{SC} \underbrace{r}_{\Pi}$$

Then, an alternate (and shorter) derivation is:

$$\dfrac{\dfrac{\dfrac{\vdots}{p \to q, p \vdash_{SC} q}\,(\text{Exc L})}{p, p \to q \vdash_{SC} q} \qquad \dfrac{\dfrac{\vdots}{q \to r, q \vdash_{SC} r}\,(\text{Exc L})}{q, q \to r \vdash_{SC} r}}{p, p \to q, q \to r \vdash_{SC} r}\,(\text{Cut})$$

Once again, the incomplete parts are similar to that of Example 1.15.

A quick scan of sequent calculus rules reveals that wffs in rule premises do not syntactically contain logical operators (they may of course have embedded logical operators not explicitly mentioned). This leads to the question of how to derive a sequent containing subformulas of a premise ϕ. The cut rule provides one elegant method for doing this, as useful conclusions may be drawn from ϕ and a cut could then be used to derive the desired sequent:

Example 1.18: Suppose we wish to derive $p \vdash_{SC} q$ from $p \vdash_{SC} q \wedge r$ Observing that q follows from $q \wedge r$, a promising approach seems to be to apply cut using $q \wedge r$ as the intermediate lemma. Given this observation, the derivation is simple:

$$\dfrac{\dfrac{\dfrac{\dfrac{p \vdash_{SC} q \wedge r}{p \vdash_{SC} q \wedge r, q}\,(\text{Wk R}) \qquad \dfrac{q \vdash_{SC} q}{q \wedge r \vdash_{SC} q}\,(\wedge \text{ L})}{p \vdash_{SC} q, q \wedge r}\,(\text{Exc R})}{p \vdash_{SC} q, q}\,(\text{Cut})}{p \vdash_{SC} q}\,(\text{Con R})$$

Although the use of cut can sometimes shorten a derivation greatly, it also causes problems. A scan of sequent calculus rules in Figure 1.6 shows that any wff appearing in the premise of a rule must be a subformula of some wff in the rule conclusion for all but the cut rule. This provides guidance in deriving sequents, since the only choice (working backwards) lies in the selection of rule to apply. In contrast, the cut rule expects the prover to determine a wff ϕ, and there is no restriction that this ϕ is a subformula or even that it is shorter than the sequent – indeed, many conceptually elegant derivations apply cut in non-obvious ways. Example 1.16 showed that creativity is needed even for cut-free derivations, but the use of cut further complicates tree construction significantly. Such creativity is obviously not automatable. This raises the question of whether a cut-free derivation is possible for all derivable sequents, which Gentzen famously answered affirmatively[10]:

Def: A sequent calculus rule R is said to be **admissible** if the existence of a derivation for R's premise/s implies that there is also a derivation of R's conclusion (without using R).

Theorem 1.5 ***(Cut Elimination Theorem)***
The cut rule for PC is admissible.

Gentzen's proof of the theorem was long, and started by essentially showing how to replace the cut wff ϕ with derivations for all possible structural forms of ϕ. By structural induction on ϕ, the theorem can then be concluded. The proof was constructive in that it showed how to produce a derivation of the rule's conclusion given the derivations of the rule's premises.

It is important to not conflate the notion of an admissible rule with that of a derived rule, which is one that can be expressed in terms of other rules. For

[10] Although his theorem was for the first order logic version of sequent calculus, its restriction to PC is stated here.

example, one possible derived rule is:

$$\frac{\phi, \psi, \phi, \Gamma \vdash_{SC} \Delta}{\phi, \psi, \Gamma \vdash_{SC} \Delta}$$

as it can be shown by applying an exchange and then a contraction. Although every derived rule is also admissible, an admissible rule R is not necessarily derived from other rules since admissibility only means that there is another R-free derivation for the same sequent. The cut rule is an example of a non-derived but admissible rule – it is clearly not a derived rule since the rule's premises may have wffs that are not subformulas of wffs in the conclusion, which is not the case for any other rules as already observed. Thus, the cut elimination theorem merely asserts that if a derivation of a sequent is given, there is another cut-free derivation. The proof of the theorem provides a means of constructing this cut-free derivation, but it does not provide a means to construct a cut-free derivation based merely on knowledge of the existence of a derivation with cut. For example, the derivation of Example 1.18 starts with $p \vdash_{SC} q \wedge r$, and a derivation for this sequent (which is of course not possible) is needed to construct a cut-free derivation of $p \vdash_{SC} q$. As a practical consequence of this distinction between derived and admissible rules, any additions to sequent calculus rules (perhaps for another logic) may invalidate proofs of cut rule admissibility, unlike the case for derived rules.

Soundness and Completeness

Section 1.3 gave an entailment relationship based on the model theoretic semantics of PC and denoted using the $\models$ symbol while this section gave a provability relation denoted using the $\vdash_{SC}$ symbol. It is obviously desirable that the two coincide – *i.e.*,

$$\phi_1, \phi_2, \ldots, \phi_m \models \psi_1, \psi_2, \ldots, \psi_n \text{ iff } \phi_1, \phi_2, \ldots, \phi_m \vdash_{SC} \psi_1, \psi_2, \ldots, \psi_n$$

and we now address this.

A proof system is said to be **sound** if everything provable in the system is in fact valid. Conversely, a proof system is said to be **complete** if it is capable of proving all valid statements.

Theorem 1.6 ***(Soundness Theorem)****:*
If $\Gamma \vdash_{SC} \Delta$ *then* $\Gamma \models \Delta$

Proof: Suppose $\Gamma \vdash_{SC} \Delta$. Then there exists some finite derivation tree of the sequent starting from axioms. The proof proceeds by structural induction over this tree.

Base Case: Tree leaves correspond to axioms, and soundness trivially follows since $\phi \models \phi$.

Inductive case: We need to show that for each rule, if the premise/s of the rule hold (the inductive hypothesis), than so do the conclusion/s.

For the cut rule, suppose $\mathcal{M}$ is a model such that:

1. If $\mathcal{M} \models \gamma$ for every $\gamma \in \Gamma$, then $\mathcal{M} \models \delta$ for some $\delta \in \Delta$ or $\mathcal{M} \models \phi$. Note that although Γ and Δ are sequences of wffs, set notation is used here and in the rest of this proof for readability. It is left as an exercise for the reader to write it properly (though with some loss of readability).
2. If $\mathcal{M} \models \phi$ and $\mathcal{M} \models \sigma$ for every $\sigma \in \Sigma$, then $\mathcal{M} \models \pi$ for some $\pi \in \Pi$.

Now suppose $\mathcal{M} \models \gamma$ and $\mathcal{M} \models \sigma$ for every $\gamma \in \Gamma$ and $\sigma \in \Sigma$. It is sufficient to show that $\mathcal{M}$ then models either δ or π (though this choice of δ,π is stronger than needed for the conclusion). From 1, $\mathcal{M}$ models δ or ϕ. In the former case, we are done; in the latter case 2 guarantees that it also models π and we are again done. The cut rule is thus sound.

Soundness of the remaining rules can be shown in similar fashion, and their [simpler] proofs are left as an exercise for the reader.
QED

The following alternate form of soundness trivially follows:
Corollary: If $\vdash_{SC} \phi$ then $\models \phi$.
Thus, to show that a wff ϕ valid, it suffices to produce a derivation tree of $\vdash_{SC} \phi$.

Theorem 1.7 ***(Completeness Theorem)****:*
If $\Gamma \models \Delta$ *then* $\Gamma \vdash_{SC} \Delta$

Proof: By Theorem 1.3, we may assume that Γ and Δ contain only the $\neg$ and $\vee$ operators. Suppose $\Gamma \models \Delta$ and let $\Gamma = \gamma_1, \gamma_2, \ldots, \gamma_m$, $\Delta = \delta_1, \delta_2, \ldots, \delta_n$. The proof proceeds by strong induction over the number of operators in $\{\Gamma, \Delta\}$.

Base Case (no operators): In this case, the γ_i's and δ_i's are all propositions, which we denote without loss of generality as p_i and q_i respectively. Moreover, every model of $\bigwedge_{i=1}^{m} p_i$ is a model of q_k for some $1 \leq k \leq n$. The only way for this to happen is that $q_k = p_{k'}$ for some $1 \leq k' \leq m$. It is then simple to construct a sequent calculus derivation of $p_1, p_2, \ldots, p_m \vdash_{SC} q_1, q_2, \ldots, q_n$, or $\Gamma \vdash_{SC} \Delta$, starting with the axiom $p_{k'} \vdash_{SC} q_k$ and using structural rules to add all other p_i's and q_i's at the appropriate places.

Inductive Hypothesis: Assume that for any Γ, Δ containing at most s total operators and such that $\Gamma \models \Delta$, $\Gamma \vdash_{SC} \Delta$.

Inductive Case: We need to show that the theorem holds for $s+1$ total operators, and there are 4 cases to consider.
In case 1, there is some $1 \leq k \leq m$ such that $\gamma_1, \ldots, \gamma_{k-1}, \neg\gamma_k, \gamma_{k+1}, \ldots, \gamma_m \models \Delta$, where $\gamma_1, \ldots, \gamma_m, \Delta$ contains at most s operators.

Consider any model $\mathcal{M}$. There are two types of models to consider.

- Restricting to models such that $\mathcal{M} \models \gamma_k$:
  The $\gamma_1, \ldots, \gamma_{k-1}, \gamma_{k+1}, \ldots, \gamma_m \models \gamma_k, \Delta$ relation holds by definition of the entailment relationship (the left side of $\models$ is irrelevant here).

- Restricting to models such that $\mathcal{M} \models \neg\gamma_k$:
  The $\gamma_1, \ldots, \gamma_{k-1}, \gamma_{k+1}, \ldots, \gamma_m \models \gamma_k, \Delta$ relation holds once again, this time since it follows from the condition for case 1 (the γ_k on the right side of $\models$ is irrelevant here).

Thus, $\gamma_1, \ldots, \gamma_{k-1}, \gamma_{k+1}, \ldots, \gamma_m \models \gamma_k, \Delta$. By the inductive hypothesis, $\gamma_1, \ldots, \gamma_{k-1}, \gamma_{k+1}, \ldots, \gamma_m \vdash_{SC} \gamma_k, \Delta$, and an application of the ($\neg$ L) rule along with at most $m-1$ exchange rules results in $\gamma_1, \ldots, \gamma_{k-1}, \neg\gamma_k, \gamma_{k+1}, \ldots, \gamma_m \vdash_{SC} \Delta$.

The remaining three cases are:

2. There is some $1 \leq k \leq m$ such that
   $\gamma_1, \ldots, \gamma_{k-1}, \gamma_k \vee \gamma_k', \gamma_{k+1}, \ldots, \gamma_m \models \Delta$
3. There is some $1 \leq k \leq n$ such that
   $\Gamma \models \delta_1, \ldots, \delta_{k-1}, \neg\delta_k, \delta_{k+1}, \ldots, \delta_n$
4. There is some $1 \leq k \leq n$ such that
   $\Gamma \models \delta_1, \ldots, \delta_{k-1}, \delta_k \vee \delta_k', \delta_{k+1}, \ldots, \delta_n$

The proofs for these cases are left as an exercise for the reader.
QED

As with soundness, an alternate form of completeness readily follows:
Corollary: If $\models \phi$ then $\vdash_{SC} \phi$.

Note that despite the soundness and completeness of sequent calculus, nothing has been said about how easy it is to find a proof, whether manually or in an automatable fashion. In fact, some examples showed the need for creativity in finding a proof. However, once a derivation is found, it is trivial to check that it is correct by merely ensuring that each step in the derivation follows from the corresponding sequent calculus rule. This check is purely syntactic and mechanical (thus automatable).

Other Classical Approaches to Proof Theoretic Semantics

Sequent calculus is but one of several classical approaches to proof theory for first order logic, which we have specialized to PC in this chapter. Two particularly prominent ones are Hilbert's calculus and natural deduction. Hilbert's calculus shares similarities with Gentzen's system but has more axioms and just two

rules, only one of which is relevant to PC. Although it is possible to translate between derivations in the two systems and they are thus equivalent, derivations in Hilbert's calculus tend to be longer and less intuitive. This is largely due to the lack of rules, resulting in the need to apply them creatively instead of following the structure of the sequent.

Natural deduction is closely related to sequent calculus and has rules similar to those of sequent calculus. Whereas a sequent calculus derivation tree derives a sequent by growing wffs/sequents going down the tree, natural deduction provides rules that control both the introduction and elimination of operators so that the desired wff can be deduced. In general, introduction rules are applied from the bottom while elimination rules are applied from the top, making the construction of derivations more bidirectional compared to the primarily bottom-up approach of sequent calculus.

As with Hilbert systems, natural deduction derivations can also be reduced to sequent calculus derivations, making the two equivalent. While some consider natural deduction simpler to use by humans, it is generally harder to provide automated support for since it is not generally clear which operators should be introduced or eliminated at an arbitrary stage of the derivation, especially since derivations are often bidirectional.

Additionally, there are also variants of the sequent calculus system presented here, many of which attempt to simplify certain common derivation patterns. If the goal is automation, it is also common to represent sequences of wffs as sets or multisets instead of sequences, thereby avoiding some structural rules.

Classical approaches to proof theories were generally not developed with automation in mind – indeed, most were developed before the invention of the modern computer. Even manual derivations are not always easy to devise. More recent approaches are designed to be algorithmic and automatable, even though they may have high complexity. Although the term "proof theory" often refers to only the classical approaches, we will use it to encompass the classical as well as automatable algorithmic approaches. The next few sections discuss the major classes of algorithmic approaches.

1.10 Semantic Tableau

Semantic tableaux were invented by the logician Evert Willem Beth in the 1950s and later simplified by the logician Raymond Smullyan. The method has since been applied to many other logics and is the most common approach for some logics. Although it shares similarities with sequent calculus, it is algorithmic and simple to automate. To prove validity of a wff ϕ, we can instead show that $\neg\phi$ is unsatisfiable due to the unsatisfiability theorem (1.1) proved earlier. The tableau procedure exploits this theorem by attempting to construct a model of $\neg\phi$. If the attempt fails, we can then conclude that ϕ is valid. It is implicit in this argument that all attempts must fail, and the failure to identify a model of $\neg\phi$ is not merely a bad attempt. Since the tableau procedure produces models for wffs, it can also be applied on a wff ϕ to check satisfiability of ϕ.

$$\frac{\neg\neg\phi}{\phi}\ (\varepsilon)$$

$$\frac{\phi_1 \wedge \phi_2}{\phi_1 \quad \phi_2}\ (\alpha) \qquad \frac{\neg(\phi_1 \vee \phi_2)}{\neg\phi_1 \quad \neg\phi_2}\ (\alpha)$$

$$\frac{\phi_1 \vee \phi_2}{\phi_1 | \phi_2}\ (\beta) \qquad \frac{\neg(\phi_1 \wedge \phi_2)}{\neg\phi_1 | \neg\phi_2}\ (\beta)$$

Figure 1.7: Tableau Rules for PC

The model construction proceeds by reducing the wff based on the syntactic structure of the formula. Figure 1.7 lists reductions for each possible syntactic form – these can informally be read as 'if a wff matches the top of the rule, it can be reduced to the bottom of the rule'.

At each step of the tableau construction, we pick a single unreduced wff ψ and determine which syntactic form it follows (either ε, α, or β). The appropriate tableau rule is then applied to ψ by creating child wffs corresponding to the bottom of that rule. For example, the ε rule can be applied to the wff $\neg\neg(p_1 \vee p_2)$ to derive a child wff $p_1 \vee p_2$, drawn as follows:

$$\frac{\neg\neg(p_1 \vee p_2)}{p_1 \vee p_2}$$

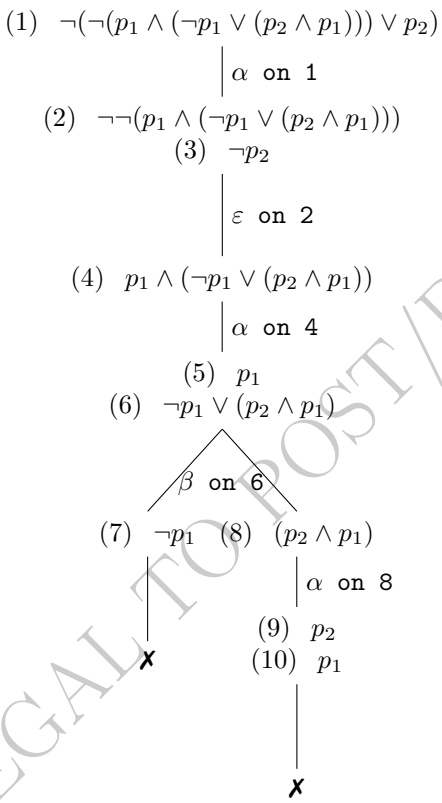
The β rules are slightly more complicated than the others since they are *branching* rules, meaning that 2 children are created, one per branch.

The tableau rules are used to construct a tree where each step follows from a single tableau rule. In PC tableaux (but not for all other logics), a wff is never expanded twice in the same branch. If at any point during the tableau construction a wff $\neg\phi$ appears as a descendant of an ancestor wff ϕ (or vice versa), that branch of the tree is terminated (marked by an ✗). This is termed as a **closed branch**, and the entire tableau is deemed closed if all branches are closed. If the tableau is closed, we conclude that the original wff $\neg\phi$ is unsatisfiable, and ϕ is thus valid due to the unsatisfiability theorem (1.1).

Example 1.19: We wish to prove that $\phi \equiv \neg(p_1 \wedge (\neg p_1 \vee (p_2 \wedge p_1))) \vee p_2$ is valid. To do this, we need to show that ϕ's negation

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

is unsatisfiable. The tableau for this (with nodes/edges labeled for reference/justification) is:



The two branches close due to the (5)-(7) and (3)-(9) contradictions (note that (7)-(10) is not a contradiction since neither is an ancestor of the other). Since all branches close, we conclude that $\neg\phi$ is unsatisfiable and ϕ is thus valid.

Although this example almost always expands a recent wff, this will not be true in general, especially for larger wffs/tableaux. In such cases, a good choice of ancestor wff to expand can dramatically affect tableau size.

Soundness, Completeness, and Decidability

Given a wff ϕ, the procedure constructs a tableau for $\neg\phi$ and concludes that ϕ is valid if the tableau closes, which we denote $\vdash_{TAB} \phi$. As in the sequent calculus,

the tableau procedure is sound and complete.

Theorem 1.8 $\vdash_{TAB} \phi$ *iff* $\models \phi$.

Proof (sketch): Consider the tableau rules of Figure 1.7. We first make the observation that for each non-branching rule, a model $\mathcal{M}$ models the premise of the rule iff it models every wff in the conclusion of the rule, as can be easily shown using the model theoretic semantics. Similarly, for each branching rule, $\mathcal{M}$ models the premise of the rule iff it models at least one of the wffs in the conclusion of the rule. Neither of these observations is surprising since the tableau rules closely mirror the model theoretic semantics.

It is then simple to apply structural induction over the tableau to show that for each structural form of wff, its models correspond to non-closing tableau paths. The tableau for $\neg\phi$ thus closes iff $\neg\phi$ is unsatisfiable. The theorem then follows from the unsatisfiability theorem.
QED

As with any induction, the proof only applies if the tableau is finite, which we now show.

Theorem 1.9 *The tableau procedure for propositional logic terminates.*

Proof: An important observation from the rules of Figure 1.7 is that all wffs generated by the tableau rules are subformulas of $\neg\phi$, possibly preceded with a single $\neg$ operator. Since there are only finitely many subformulas (even if a $\neg$ can be added to their left) and the same subformula is never expanded twice on the same branch, every branch must eventually terminate, either with a ✗ or with all nodes expanded. The number of branches is bounded by [an exponential function of] the number of β-subformulas, which is also finite. Since both the number of branches and the length of each branch is finite, the tableau must itself be finite, and the theorem follows.
QED

Considerations in Tableau Construction

Although the tableau procedure terminates, tableaux may be large as it is simple to construct wffs for which the number of tree branches is proportional to the number of total models. In general, the size of a tableau is proportional to the size of the wff being considered. It is also possible to generate different tableaux for the same wff since our procedure only gave a set of rules that may be applied, and it did not specify which rule to apply if there is a choice of wffs to expand. However, only one closed tableau needs to be constructed to show unsatisfiability.

It is obviously desirable to construct a smaller tableau, and a natural heuristic for producing smaller tableaux is to preferentially select for expansion any wff that would result in closing that branch. Beyond that, since branching rules

(*i.e.*, β) are clearly the culprit in tableau size, we normally apply the ε and α rules first, delaying β until no more non-branching wffs are available to expand. For example, suppose there are two unexpanded wffs: $p_1 \wedge p_2$, $p_3 \vee p_4$. In that case, expansion of $p_1 \wedge p_2$ first avoids the expansion of $p_3 \vee p_4$ in any branch whose closing does not need its expansion. Such a choice effectively captures multiple models in the same branch, potentially capturing many possible models of the wff in the same branch. There are other methods for systematic construction of compact tableaux, but these are beyond the scope of this text.

As alluded to above, all wffs appearing in a tableau need not be expanded, namely in the case where the tableau closes without expansion of these wffs. Expansion of these irrelevant wffs only serves to expand the size of the tableau (see Exercise **??**), and should thus be avoided though it will not in general be possible to determine their irrelevance.

One minor issue remains – the tableau procedure presented here assumes for simplicity that constants, and the $\rightarrow$ and $\leftrightarrow$ operators have been eliminated. This preprocessing may not be advisable, and the avoidance of such preprocessing is in fact one advantage of the tableau method to avoid normal forms potentially leading to much larger wffs. It is of course simple enough to add tableau rules for these operators or even rules for various special forms of wffs, though the latter needs care to maintain soundness / completeness / decidability, especially for logics more complicated than PC. An important consideration in designing tableau algorithms is the tradeoff between the additional time needed to check for these special forms of wffs (for relatively few cases) vs. the reduced time when the special forms are applicable. Similarly, additional special forms of wffs (and their negations) that can lead to early closing of branches may also be identified.

1.11 Resolution

One of the most common automatable proof procedures for the PC validity problem is resolution, along with the Davis-Putnam-Logemann-Loveland (DPLL) extension which will be discussed in the next section. Resolution is also a refutation method like tableaux – that is, validity of the wff ϕ is shown by showing that $\neg\phi$ is unsatisfiable, applying the Unsatisfiability Theorem (1.1). Unlike tableaux, we assume formulas have first been converted to a normal form (CNF), thus avoiding the need for rules tailored to each syntactic form; in fact, only one rule will be needed. Of course, the tradeoff is that the CNF form might be significantly larger. However, the use of a single rule can simplify the generation of derivations compared to tableaux, especially if intended for automation.

Recall that each conjunct in a CNF wff is called a clause; thus, each clause is a disjunction of literals. To prove $\models \phi$, we convert $\neg\phi$ to CNF, and represent it in a **clausal form**. The clausal form of a clause is a set containing the literals of the clause, while the clausal form of a CNF wff is a set of all the clauses. For example, the clausal form of the wff $(p_1 \vee p_3 \vee \neg p_7) \wedge (p_2 \vee p_1)$

is $\{\{p_1, p_3, \neg p_7\}, \{p_1, p_2\}\}$. Clausal form is used so that the properties of sets can be assumed – in particular, the same literal can not occur twice in a set by definition (though ℓ and $\neg\ell$ can both appear), and there is no implicit ordering of elements in sets. The following resolution rule on two clauses is then applied repeatedly until $\perp$ (*i.e.*, the empty clause) is derived:

$$\frac{\{\ell_{11}, \ell_{12}, \ldots, \ell_*, \ldots, \ell_{1m}\} \quad \{\ell_{21}, \ell_{22}, \ldots, \neg\ell_*, \ldots, \ell_{2n}\}}{(\{\ell_{11}, \ldots, \ell_{1m}\} \backslash \{\ell_*\}) \cup (\{\ell_{21}, \ldots, \ell_{2n}\} \backslash \{\neg\ell_*\})}$$

where the ℓ's are literals. This rule applies to two clauses that have literals ℓ_* and $\neg\ell_*$ respectively and produces a new clause containing all other literals in the two clauses. The bottom of the rule is called the **resolvent**, and the two clauses on the top are said to **clash** due to the clashing literal ℓ_*. The resolvent of clauses C_1 and C_2 is denoted $res(C_1, C_2)$, or $res_p(C_1, C_2)$ when needed to signify that the particular resolution being done is based on clashing p (since the clauses may also have other clashing literals). The definitions of satisfiability and validity are extended to clause sets in the natural way – *e.g.*, S is said to be satisfiable if the conjunction of clauses in S is satisfiable, and similarly for validity.

Example 1.20:
The resolvent of the clauses $\{\neg p_1, p_2, p_7\}$ and $\{\neg p_2, p_3\}$, with clashing literal p_2, is $\{\neg p_1, p_3, p_7\}$.
This may be written as $res_{p_2}(\{\neg p_1, p_2, p_7\}, \{\neg p_2, p_3\}) = \{\neg p_1, p_3, p_7\}$

The resolvent of the clauses $\{p\}$ and $\{\neg p, q\}$ is $\{q\}$. Note that this bears a striking resemblance to the modus ponens argument if the second clause is written as $p \rightarrow q$ and implication is conflated with provability.

The resolvent of the clauses $\{p\}$ and $\{\neg p\}$ is the empty clause $\{\}$, signifying a clause equivalent to $\perp$. Since there is no model satisfying both input clauses (*i.e.*, they are contradictory), this is as expected.

The resolvent of the clauses $C_1 \equiv \{p, q_1, \neg p\}$ and $C_2 \equiv \{\neg p, q_2\}$ is $\{q_1, \neg p, q_2\}$. Note that the $\neg p$ in C_1 is not eliminated. Also note that the resolvent is not the strongest possible inference, but this will be sometimes the case with resolution.

The resolution rule does not allow for clashing multiple literals in one step. For example, it is not allowable to resolve $\{p_1, p_2\}$ and $\{\neg p_1, \neg p_2\}$ attaining the resolvent $\{\}$. It is clear that this inference is invalid since there exist two models of the two clauses: 1) p_1 true and p_2 false, and 2) p_1 false and p_2 true. A proper application of the resolution rule could yield either $\{p_2, \neg p_2\}$ based on the p_1 clash or $\{p_1, \neg p_1\}$ based on the p_2 clash. In either case, the resulting clause is satisfiable as expected.

The resolution rule is sound since if ℓ_* is true, $\neg\ell_*$ must be false and one of the ℓ_{2k}'s must be true for the rule premise to hold. Similarly, if ℓ_* is false, one of the ℓ_{1k}'s must be true. Given that ℓ_* is either true or false, the resolvent follows. Note however that although the resolvent can be inferred from the rule

premises, the resolvent is not equivalent to the conjunction of the premises. For example, we can infer neither $\ell \vee \phi_1$ nor $\neg\ell \vee \phi_2$ from $\phi_1 \vee \phi_2$ for an ℓ that does not appear in ϕ_1 or ϕ_2.

The impact of the resolution rule is that it allows for the elimination of a literal from two clauses, and it can be applied repeatedly to eliminate all literals until $\bot$ is inferred. More formally, the resolution proof procedure to prove validity of ϕ is:

1. Convert $\neg\phi$ to CNF and express in clausal form.

2. Repeat until $\bot$ is derived (*i.e.*, a clause is empty) or no unresolved clashing pair of clauses exists:

   (a) Pick 2 clauses that clash.

   (b) Resolve them, and add their resolvent. The use of clausal notation implicitly prevents addition of the resolvent if it already exists as another clause.

3. If $\bot$ has been proven, conclude that $\neg\phi$ is unsatisfiable, and ϕ is thus valid. Otherwise (no more unresolved clashing clauses exist), conclude that $\neg\phi$ is satisfiable, and ϕ is thus not valid.

Example 1.21: We solve a complete problem by resolution. Consider the statements:

1. If it rains, I brought an umbrella
2. If I brought an umbrella, I do not get wet
3. If it does not rain, I do not get wet

where we want to prove that I do not get wet. Selecting atoms r, u, and w representing "It rains", "I brought an umbrella", and "I get wet" respectively, the three statements are formalized as $\phi_1 \equiv r \rightarrow u$, $\phi_2 \equiv u \rightarrow \neg w$, and $\phi_3 \equiv \neg r \rightarrow \neg w$ respectively. We wish to prove validity of $(\phi_1 \wedge \phi_2 \wedge \phi_3) \rightarrow \neg w$, or equivalently that its negation $\phi_1 \wedge \phi_2 \wedge \phi_3 \wedge \neg\neg w$ is unsatisfiable.
We first convert to CNF clausal form, resulting in: $\{\{\neg r, u\}, \{\neg u, \neg w\}, \{r, \neg w\}, \{w\}\}$. One resolution proof of this is:

(1.1)	$\{\neg r, u\}$	$(Premise)$
(1.2)	$\{\neg u, \neg w\}$	$(Premise)$
(1.3)	$\{r, \neg w\}$	$(Premise)$
(1.4)	$\{w\}$	$(Premise)$
(1.5)	$\{r\}$	$(Resolve\ 1.3, 1.4)$
(1.6)	$\{\neg u\}$	$(Resolve\ 1.2, 1.4)$
(1.7)	$\{\neg r\}$	$(Resolve\ 1.1, 1.6)$
(1.8)	$\{\}$	$(Resolve\ 1.5, 1.7)$

This proof may also be displayed graphically, as in Figure 1.8.

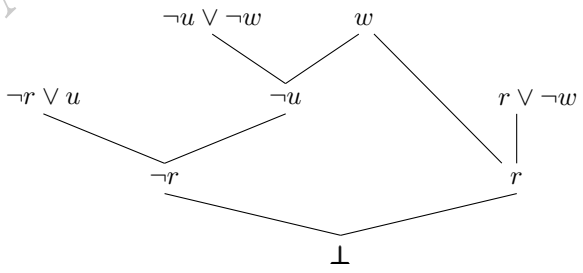


Figure 1.8: Graphical Representation of Resolution Proof

A scan of the example reveals several interesting observations. First, the same clause may be resolved multiple times – for example 1.4 needed to be resolved against two different clauses. Also, the resolution procedure does not

tell us *which* clauses to resolve, and there is almost always a choice – for example, step 1.5 may have chosen to resolve 1.1 and 1.2 instead (among other choices), or the order of steps 1.5 and 1.6 could have been swapped. Finally, the proof can be drawn as a binary tree with a root $\bot$ at the bottom, provided copies are made of multiply-used clauses (only w in the above example). Thus, we will refer to such proofs as **resolution trees**.

Soundness, Completeness, and Decidability

Given a CNF wff ϕ, suppose it is possible to infer an empty clause from $\neg\phi$. Then the resolution algorithm concludes that ϕ is valid, denoted $\vdash_{RES} \phi$. We earlier argued that the resolution rule was sound, and we formalize it with the following definition/theorem:

Def: Two wffs are **equisatisfiable** if they are both satisfiable or both unsatisfiable.

Theorem 1.10 ***(Equisatisfiability)****: The conclusion of the resolution rule is equisatisfiable to the conjunction of its premises.*

Proof: The resolution rule can be interpreted as:

$$\frac{\phi \vee p \quad \psi \vee \neg p}{\phi \vee \psi}$$

where ϕ and ψ are disjunctions of literals excluding p and $\neg p$ respectively.

Case 1 (unsatisfiable): Suppose $\phi \vee \psi$ is unsatisfiable. Then, both ϕ and ψ are unsatisfiable. Thus, if $\phi \vee p$ is satisfiable, it must be in a model assigning $\top$ to p. Similarly, for $\psi \vee \neg p$ to be satisfiable, it must be in a model assigning $\bot$ to p. Since these two models are contradictory, the conjunction of the premises must be unsatisfiable.

Case 2 (satisfiable): Suppose $\phi \vee \psi$ is satisfiable. Then [at least] one of ϕ or ψ is satisfiable. If ϕ is satisfiable, pick any model of ϕ and extend it to assign $\bot$ to p (this can be done since ϕ does not contain the literal p). This extended model clearly models both $\phi \vee p$ and $\psi \vee \neg p$. Similarly, p can be assigned $\top$ in the case where ψ is satisfiable (either assignment to p can be made if both ϕ and ψ are satisfiable). Thus the conjunction of the premises is satisfiable in all cases.
QED

Equivalently, the resolution rule preserves both satisfiability and unsatisfiability.

Note that two equisatisfiable wffs do not necessarily have the same models – for example, in the above version of the resolution rule, the conclusion may have a model satisfying all of ϕ, $\neg\psi$, and p, but this model does not satisfy $\psi \vee \neg p$ (and thus not the conjunction of the premises). However, the following properties trivially hold, where C_1 and C_2 are clashing CNF clauses, and $\mathcal{M}$ some model:

P1: If $\mathcal{M} \models res(C_1, C_2)$, then $\mathcal{M} \models C_1$ or $\mathcal{M} \models C_2$.

P1': If $C_1 \vee C_2$ is unsatisfiable, $res(C_1, C_2)$ is also unsatisfiable.

P2: If $\mathcal{M} \models C_1$ and $\mathcal{M} \models C_2$ then $\mathcal{M} \models res(C_1, C_2)$.

P2': If $res(C_1, C_2)$ is unsatisfiable, the conjunction of C_1 and C_2 is also unsatisfiable.

Note that P1 is stronger than needed as the conclusion may hold even if $\mathcal{M} \not\models res(C_1, C_2)$. Note also that P1' and P2' are equivalent to their unprimed versions P1 and P2 respectively. These equivalences will be trivial for the reader familiar with first order logic (Chapter 3) since the primed properties are just the contrapositives of their unprimed versions, and contrapositives will be shown to be equivalent. But since the properties are about some/all models, they are only expressible in first order logic and not PC, thus requiring a small amount of explicit reasoning for now.

Soundness of the resolution algorithm readily follows.

Theorem 1.11 *If* $\vdash_{RES} \phi$, *then* $\models \phi$.

Proof: Suppose $\vdash_{RES} \phi$. Then, there is a resolution derivation from $\neg\phi$ to an empty clause. Consider the set of clauses S of $\neg\phi$ in the resolution tree leading to the empty clause in this proof, and suppose S is satisfied by some model $\mathcal{M}$. An induction applying property P2 shows that $\mathcal{M}$ must also satisfy every clause in this tree, leading to a contradiction since the empty clause is also in this tree. Thus, S must be unsatisfiable, implying that the set of all clauses in $\neg\phi$ is also unsatisfiable. It follows from the unsatisfiability theorem (1.1) that $\models \phi$.
QED

Completeness of the resolution algorithm is somewhat harder to prove, and this is as expected given the difference between properties P1 and P2. While it may seem at first glance that a structural induction using the equisatisfiability theorem (1.10) suffices, the problem is that models satisfying different paths down a resolution tree will not necessarily be the same.

Theorem 1.12 *If* $\models \phi$, *then* $\vdash_{RES} \phi$.

Proof: Denoting $\neg\phi$ as ψ, we will prove the equivalent statement that if there is no resolution derivation from ψ resulting in an empty clause, then ψ is satisfiable. We first assume that the clauses in ψ have been preprocessed to remove any clause containing both an atom p and its negation $\neg p$ – this can safely be done since such clauses are satisfied by any model and their presence thus does not affect the satisfiability of a CNF clause ψ (See also Exercise **??** for a related problem).
We will use strong induction over $k = |\mathcal{A}|$.
Inductive Hypothesis: If $|\mathcal{A}| < k$ and there is no resolution derivation starting

from ψ and leading to an empty clause, then ψ is satisfiable.
Inductive step: Suppose $\mathcal{A} = \{p_1, \ldots, p_k\}$ and there is no resolution derivation starting from ψ and leading to an empty clause. Consider a resolution tree starting from the clauses in ψ and in which no more resolutions are possible. Denoting the set of clauses in the tree S_k, remove all occurrences of p_k (including its negation) and partition the resulting clauses into three sets:

$$S_k^{\top} = \{C \backslash \{p_k\} : C \in S_k, p_k \in C\}$$

$$S_k^{\bot} = \{C \backslash \{\neg p_k\} : C \in S_k, \neg p_k \in C\}$$

$$S_k^{-} = \{C \in S_k : p_k \notin C, \neg p_k \notin C\}$$

These sets correspond to the clauses that contained p_k, those that contained $\neg p_k$, and those in which p_k was not present, respectively. Since S_k^- has fewer than k atoms and no more possible resolutions, the inductive hypothesis ensures that S_k^- is satisfiable, say by model $\mathcal{M}$.
Suppose $\mathcal{M} \models S_k^{\top}$. The model extending $\mathcal{M}$ by assigning $\bot$ to p_k satisfies every clause in S_k, and thus all the clauses in ψ, making ψ satisfiable.
Thus, we only need to consider the case where $\mathcal{M} \not\models S_k^{\top}$. Equivalently, there is some clause $C_{\top}$ such that $C_{\top} \in S_k^{\top}$ and $\mathcal{M} \not\models C_{\top}$. A similar argument can be made for some clause $C_{\bot}$ such that $C_{\bot} \in S_k^{\bot}$ and $\mathcal{M} \not\models C_{\bot}$.
S_k contains all possible resolvents, including $res_{p_k}(C_{\top} \cup \{p_k\}, C_{\bot} \cup \{\neg p_k\}) \in S_k^-$, which $\mathcal{M}$ satisfies. But property P1 shows that this resolvent is not satisfied by $\mathcal{M}$, leading to a contradiction.
QED

While the completeness theorem above may seem at first glance to be the same as for sequent calculus, it has significant differences masked by notation. Recall that $\vdash_{RES} \phi$ signifies only that there is a proof starting with the clauses of $\neg\phi$ and leading to an empty clause. This form of completeness is referred to as **refutation completeness**, in contrast with the more general form of completeness from earlier. While the more general form allows for us to derive all wffs ψ entailed by a given wff ϕ, a refutation complete system only allows us to prove that a given set of CNF wffs is unsatisfiable, with the most common application being that the wff $(\phi_1 \wedge \phi_2 \wedge \ldots \wedge \phi_n) \rightarrow \psi$ is valid (utilizing the unsatisfiability theorem). While this may seem to be a shortcoming of resolution, it is what enables an efficient (or even terminating) procedure by avoiding the derivation of wffs irrelevant to the validity of the desired wff.

From an algorithmic perspective, an important observation is that a resolution inference can not produce any new literals. Since there are only finitely many atoms (thus, literals), there are finitely many possible clauses by virtue of the set notation. Since the implementation is prevented from adding resolvent clauses that already exist, again by virtue of the set notation, this means that the procedure must eventually run out of clause pairs to resolve yielding the following theorem:

Theorem 1.13 *The resolution procedure for propositional logic terminates.*

Thus, the only possible outcomes of the algorithm are that it ends with an empty clause or that no more resolutions are possible, corresponding to the cases where $\neg\phi$ is unsatisfiable and satisfiable, respectively.

Although the resolution procedure is guaranteed to terminate, the proof need not be efficient since the algorithm does not provide a means to avoid fruitless resolution paths.

Example 1.22: Consider the set of wffs (in non-clausal form for illustrative purposes):

$$p_1 \vee p_2$$
$$\neg p_2 \vee p_3$$
$$\ldots$$
$$\neg p_{k-1} \vee p_k$$
$$\neg p_1$$
$$\neg p_2$$

where k is a large number. A quick resolution proof would simply resolve $p_1 \vee p_2$ and $\neg p_1$ to result in p_2, and then resolve that against $\neg p_2$ to infer $\perp$. However, an alternate resolution proof might apply resolution on the $k-1$ disjunctive clauses to produce the fruitless $p_1 \vee p_k$, before making any progress. For large k, this is certainly not practical!

The problem occurred since the resolution algorithm does not direct the choice of clauses to resolve. Although the problem can not be prevented in all cases, two approaches suggest themselves:

1. Restrict the form of the clauses to enable a search that avoids wasteful inferences.
2. Use heuristics to decide which clauses to resolve first and/or which clauses may be ignored.

We now discuss these two approaches, using Horn resolution and DPLL respectively. While there are resolution heuristics unrelated to DPLL, the reader is referred to [**?**] as this book will only discuss DPLL.

1.11.1 Horn Resolution

Recall that a Horn formula is a CNF formula where each clause has at most one positive literal. Two key observations are:

1. The resolvent of two Horn clauses ϕ_1 and ϕ_2 is itself a Horn clause. This occurs since there are at most two positive literals of concern (one from each of ϕ_1, ϕ_2), and one of them must have been eliminated by the resolution.

2. If a clause G with no positive literals (called a **goal**) is resolved against another clause C, the resolvent also has no positive literals since the only possible clash would have been against C's sole positive literal which the resolution removed. More succinctly, resolution against a goal produces other goals (often called subgoals).

Based on these observations, it is clear that if there is a resolution proof for a problem, it can be found by starting with a goal and repeatedly resolving against the goal to replace it with subgoals.

Example 1.23: Consider the example started in Section 1.7, augmented with a $\{getwet, \neg snow\}$ clause. Starting with the goal 1.6, we apply resolution as follows:

(1.1)	$\{rain\}$	premise
(1.2)	$\{running\}$	premise
(1.3)	$\{getwet, \neg snow\}$	premise
(1.4)	$\{getwet, \neg rain, \neg outside\}$	premise
(1.5)	$\{outside, \neg running\}$	premise
(1.6)	$\{\neg getwet\}$	goal
(1.7)	$\{\neg rain, \neg outside\}$	Resolve against 1.4
(1.8)	$\{\neg outside\}$	Resolve against 1.1
(1.9)	$\{\neg running\}$	Resolve against 1.5
(1.10)	$\{\}$	Resolve against 1.2

We have thus shown that $\neg getwet$ leads to a contradiction, and $getwet$ must thus hold.

As mentioned above, at each step of the proof, only the first goal in the set of current goals is resolved against.

While this example shows how to perform a chain of resolutions in a goal-directed manner if a derivation exists, that does not imply that any possible such chain of resolutions will be successful in deriving the empty clause. In general, there will be multiple clauses to resolve the current goal against. For example, Step 1.7 of Example 1.23 might instead resolve against 1.3, resulting in $\{\neg snow\}$ which is a deadend in the resolution derivation. Fortunately there are algorithmic solutions for Horn Clause resolution. The two most common approaches differ on whether they go backwards from the goal as above or forward from the initial clauses, and they are thus termed backward chained and forward chained respectively.

Backward Chained Horn Resolution

The backward chained approach starts with the goal as in Example 1.23, and essentially backtracks and tries alternate clauses to resolve against if a deadend

is reached:

1. Initialize S to the list of goals in the Horn formula, and let ϕ be the remaining clauses. Note that there is an implicit conjunction between goals in S.

2. Repeat until S is empty:

   (a) Let G be the first goal in S.

   (b) Find a clause $C \in \phi$ that clashes with G (skip to 2d if no such C exists). It is simple to determine the candidate clashing clauses since the goal contains only negative literals, and we only need to check the single positive literal in each other clause.

   (c) Update S to replace G with the subgoals in $res(C, G)$ (just remove G if the resolvent is empty), and skip to Step 2a next.

   (d) If unable to resolve any further, backtrack to the most recently chosen clause in 2b for which a different clashing clause exists, and retry the step with that clause. Note that this clause may be several iterations back. If no more backtracks are possible, exit loop.

3. If S is empty return "Unsatisfiable"; otherwise, return "Satisfiable". As usual, unsatisfiability indicates that the original list of [conjuncted] goals follows from the Horn formula.

The algorithm presented here is goal directed and provides an algorithmic method for exploring the search tree. This algorithm is actually just one type of backward chained technique. In particular, it relies on backtracking if unfruitful decisions are taken, while other algorithms may explore multiple paths at decision points (readers familiar with search algorithms may recognize this to be a major difference between depth-first and breadth-first search). This algorithm also affords various implementation choices that affect its behavior. First, the goal list S is often organized as a first-in-last-out data structure (*i.e.*, stack with pop and push operations for adding and removing clauses) to select the clause at steps 2a and 2c respectively, but other possibilities exist. A second major choice is in step 2b, which is typically dictated by a top-to-bottom ordering of the clauses. However, other orderings can be used, resulting in yet more backward chained algorithms.

Example 1.24: Consider the Horn formula:

$$\{\underbrace{\{p1, \neg p2, \neg p3\}}_{C_1}, \underbrace{\{p2\}}_{C_2}, \underbrace{\{p3, \neg p4, \neg p5\}}_{C_3}, \underbrace{\{p3, \neg p4, \neg p6\}}_{C_4}, \underbrace{\{p4\}}_{C_5}, \underbrace{\{p6, \neg p7\}}_{C_6}, \underbrace{\{p6, \neg p8\}}_{C_7}, \underbrace{\{p8\}}_{C_8}\}$$

with the goal list $\neg p2, \neg p1$.
The following table traces the value of S after each iteration.

	G	S	Justification
0		$\{\neg p2, \neg p1\}$	Initialization
1	$\neg p2$	$\{\neg p1\}$	Resolve against C_2
2	$\neg p1$	$\{\neg p2, \neg p3\}$	Resolve against C_1
3	$\neg p2$	$\{\neg p3\}$	Resolve against C_2
4	$\neg p3$	$\{\neg p4, \neg p5\}$	Resolve against C_3
5	$\neg p4$	$\{\neg p5\}$	Resolve against C_5
6	$\neg p5$		No clashing clause found
7	$\neg p3$	$\{\neg p4, \neg p6\}$	Backtrack to 4, Resolve against C_4
8	$\neg p4$	$\{\neg p6\}$	Resolve against C_5
9	$\neg p6$	$\{\neg p7\}$	Resolve against C_6
10	$\neg p7$		No clashing clause found
11	$\neg p6$	$\{\neg p8\}$	Backtrack to 9, Resolve against C_7
12	$\neg p8$	$\{\}$	Resolve against C_8

The negation of the goals (*i.e.*, the wff $p_2 \vee p_1$) thus follows from the remaining clauses.

The Horn restriction enables more efficient goal-directed search, though there is still some inefficiency due to the need for maintaining appropriate data structures (*e.g.*, a stack) to support backtracking – for example an efficient implementation may have marked step 4 of Example 1.24 (but not step 5, which has no alternate choices) to indicate that other choices exist to support the backtracking of step 7. Section 3.9 will show an alternate interpretation of Horn Clauses where each clause has a procedural semantics, and this interpretation forms the basis of a class of programming languages. The most common of these languages, Prolog, for the first order logic version of backward chained Horn resolution, makes precisely those choices alluded to before Example 1.24.

Although the backward chained algorithm provides an algorithmic way of exploring the search tree in a goal driven manner, it suffers from several issues. First, it does not prevent duplicate clauses from being generated and thus may not terminate. For example, resolving the goal $\neg p$ against the wff $\{\{p, \neg p\}\}$ results in a subgoal $\neg p$, leading to an infinite loop. This can be generalized to any Horn formula in which the clauses lead to circularities – for example $\{\{p_1, \neg p_2\}, \{p_2, \neg p_3\}, \ldots, \{p_{n-1}, \neg p_n\}, \{p_n, \neg p_1\}\}$ with the goal $\neg p_1$. Although a check for duplicates can be done, that check at every resolution step would detract from efficiency (though Horn clause applications often do not exhibit such forms in practice). Second, a goal-driven approach may not always be the most efficient way of exploring the search space, and a forward chained version

is more appropriate in these cases.

Forward Chained Horn Resolution

Whereas the previous algorithm performed Horn resolution in a goal-directed manner, there are also efficient algorithms going forward. A common algorithm functions by marking subformulas as their satisfiability is determined, working up in complexity. For example, if a CNF wff contains the clauses $\{r\}$, $\{q, \neg r\}$, and $\{p, \neg q, \neg r\}$, a satisfying model must assign T to r after which it is easy to see that q must also hold based on the second clause, and p must then also hold based on the third clause. More generally, given a Horn formula ϕ in clausal notation:

1. For every clause in ϕ that has exactly one positive literal p and no negative literals, mark every occurrence of p in ϕ.

2. Repeat until no such clause exists:

   (a) If there is a clause of the form $\{\neg q_1, \neg q_2, \ldots, \neg q_n\}$ in which all the q_i's have already been marked, return "unsatisfiable".

   (b) For every clause of the form $\{\neg q_1, \neg q_2, \ldots, \neg q_n, p\}$ such that all the q_i's have already been marked (and p is not already marked), mark every occurrence of p in ϕ.

3. Return "satisfiable". The model is given by the marking, where all marked atoms are true in the model.

As usual, a return value of "unsatisfiable" indicates that the negation of the initial goal/s together with the other clauses is unsatisfiable, or equivalently that the goals are implied by the remaining clauses.

It is easy to see that the forward chained algorithm terminates since each nonterminating iteration in Step 2 marks all occurrences of some atom and there are only finitely many atoms. For the same reason, the complexity of this algorithm is linear since it is at worst proportional to the number of atoms.

Example 1.25: Consider Example 1.24 again. The following table lists the wff after each step of the algorithm, where atoms are grayed out to indicate that they are marked:

Step	**Wff**
Initial	$\{\{p1, \neg p2, \neg p3\}, \{p2\}, \{p3, \neg p4, \neg p5\}, \{p3, \neg p4, \neg p6\}, \{p4\}, \{p6, \neg p7\}, \{p6, \neg p8\}, \{p8\}, \{\neg p1, \neg p2\}\}$
1	$\{\{p1, \neg p2, \neg p3\}, \{p2\}, \{p3, \neg p4, \neg p5\}, \{p3, \neg p4, \neg p6\}, \{p4\}, \{p6, \neg p7\}, \{p6, \neg p8\}, \{p8\}, \{\neg p1, \neg p2\}\}$
2a	Not applicable
2b	$\{\{p1, \neg p2, \neg p3\}, \{p2\}, \{p3, \neg p4, \neg p5\}, \{p3, \neg p4, \neg p6\}, \{p4\}, \{p6, \neg p7\}, \{p6, \neg p8\}, \{p8\}, \{\neg p1, \neg p2\}\}$
2a	Not applicable
2b	$\{\{p1, \neg p2, \neg p3\}, \{p2\}, \{p3, \neg p4, \neg p5\}, \{p3, \neg p4, \neg p6\}, \{p4\}, \{p6, \neg p7\}, \{p6, \neg p8\}, \{p8\}, \{\neg p1, \neg p2\}\}$
2a	Not applicable
2b	$\{\{p1, \neg p2, \neg p3\}, \{p2\}, \{p3, \neg p4, \neg p5\}, \{p3, \neg p4, \neg p6\}, \{p4\}, \{p6, \neg p7\}, \{p6, \neg p8\}, \{p8\}, \{\neg p1, \neg p2\}\}$
2a	Abort (not satisfiable)

Since the set of clauses is unsatisfiable, $p1 \vee p2$ follows from the remaining clauses, as usual.

There are obvious tradeoffs in picking between the backward and forward chained algorithms. The backward chained algorithm avoids reasoning about clauses irrelevant to the goal, which is especially advantageous with a large database of rules. However, a direct implementation may lead to nontermination, and its backtracking approach imposes implementation complications as outlined earlier. The forward chained algorithm is the opposite, and is thus preferable when there is no reason to expect many clauses irrelevant to the proof. It is also preferable if multiple runs are to be made (with different goals) since resolutions common to runs need only be made once. Artificial intelligence reasoning systems sometimes incorporate the two techniques in the hope of exploiting the advantages of both.

1.12 DPLL

In the general CNF case where all clauses need not be Horn, heuristics are needed to address the issues described in the previous section. An observation of typical resolution proofs yields two obvious heuristics:

UNIT Propagation: If a clause C has just one literal ℓ (called a unit clause), ℓ must be true for the CNF formula to hold. Thus, we can simply eliminate C and assign $\top$ or $\bot$ to the corresponding atom depending on whether ℓ is positive

or negative. Any other clauses containing ℓ are deleted (since we have made the literal true in a disjunctive clause). For any other clause containing the negation of ℓ, the clause remains but with ℓ's negation deleted (since we have made it false). If the clause becomes empty, we have shown that the clause, and thus the CNF wff, is unsatisfiable. The impact of this rule is to eliminate all references to an atom from the CNF wff. The rule can also be seen as a bulk resolution of the unit clause against *all* other clashing clauses.

Example 1.26: Consider the wff

$$(p_1 \vee p_2) \wedge (\neg p_2 \vee p_3 \vee p_4) \wedge (p_1 \vee p_4) \wedge p_2$$

It contains the unit clause p_2, and applying UNIT on this results in the much shorter wff

$$(p_3 \vee p_4) \wedge (p_1 \vee p_4)$$

PURE Literal Elimination: A pure literal ℓ is one which occurs only positively or only negatively in the CNF formula. In that case, it is not possible to resolve based on ℓ, and we can simply delete all clauses containing ℓ, implicitly making an assignment of $\top$ to ℓ. As this rule is merely the elimination of clauses irrelevant to any resolution proof, the unit and pure rules together effectively do mass resolution to quickly reduce problem size.

Example 1.27: Consider the wff

$$\phi \equiv (p_1 \vee p_2) \wedge (\neg p_2 \vee p_3 \vee \neg p_4) \wedge (\neg p_1 \vee \neg p_4) \wedge p_2$$

It contains a pure literal $\neg p_4$, and applying PURE results in

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

Note that it can not be concluded in the above example that all models of ϕ assign $\bot$ to p_4, as the pure literal rule only says that this assignment does not affect the satisfiability of ϕ. Indeed, the model $\{p_2, p_3, p_4\}$ satisfies ϕ. Recalling that the resolution rule only guarantees equisatisfiability, this should not be surprising for any extension of resolution.

The mathematician Martin Davis and philosopher Hilary Putnam created an algorithm that exploits these rules, and it was later extended by Davis, George Logemann and David W. Loveland, resulting in what is commonly known as the DPLL algorithm. Given a CNF formula ϕ, the algorithm is:

function DPLL(ϕ)
// Precondition: ϕ is a CNF wff
// Postcondition: the return value is $\top$ iff ϕ is satisfiable
  Apply UNIT as much as possible
  Apply PURE as much as possible
  if ϕ is empty **then return** $\top$
  else if Some clause in ϕ is empty **then return** $\bot$
  else // SPLIT Rule
    Select a literal ℓ in ϕ
    return DPLL($\phi \wedge \ell$) $\vee$ DPLL($\phi \wedge \neg\ell$)
  end if
end function

The third case above is called the splitting rule since it splits the problem of determining whether ϕ is satisfiable into one of determining whether ϕ is satisfiable when ℓ is true, and one where it is satisfiable when ℓ is false. It may be optimized so that if one of the two split cases is found satisfiable, the other one is not tried.

It is simple to see that the DPLL algorithm terminates since each step eliminates at least one atom: either one atom per application of UNIT and PURE, or the atom corresponding to ℓ if the splitting rule is applied. Since there are finitely many atoms, it must eventually terminate.

The DPLL algorithm can still suffer from exponential time due to the splitting rule. However, empirical evidence shows that for many real problems the bulk of the time is spent in unit reduction which quickly reduces the number of atoms (and many containing clauses), while the PURE rule serves to quickly eliminate many clauses when applicable. Note also that it is simple to modify the algorithm so that it returns a satisfying model instead of just the solution to the decision problem.

Example 1.28: We wish to test satisfiability of:

$$\{\{p_1, p_2, p_4, \neg p_8\}, \{\neg p_1, p_3, p_4\}, \{p_1, p_2, \neg p_7\}, \{p_2, \neg p_3, p_4\}, \{p_2, \neg p_4, p_5\},$$
$$\{\neg p_2, \neg p_8\}, \{p_3, \neg p_4, \neg p_7\}, \{\neg p_4, p_6\}, \{\neg p_6, p_8\}, \{\neg p_5, \neg p_6, \neg p_8\}, \{p_8\}\}$$

There is just one unit clause here ($\{p_8\}$), and applying UNIT results in:

$$\{\{p_1, p_2, p_4\}, \{\neg p_1, p_3, p_4\}, \{p_1, p_2, \neg p_7\}, \{p_2, \neg p_3, p_4\}, \{p_2, \neg p_4, p_5\},$$
$$\{\neg p_2\}, \{p_3, \neg p_4, \neg p_7\}, \{\neg p_4, p_6\}, \{\neg p_5, \neg p_6\}\}$$

This has created another unit clause ($\{\neg p_2\}$), so we apply UNIT again:

$$\{\{p_1, p_4\}, \{\neg p_1, p_3, p_4\}, \{p_1, \neg p_7\}, \{\neg p_3, p_4\}, \{\neg p_4, p_5\},$$
$$\{p_3, \neg p_4, \neg p_7\}, \{\neg p_4, p_6\}, \{\neg p_5, \neg p_6\}\}$$

There are no remaining unit clauses, but $\neg p_7$ is a pure literal, so the PURE rule produces:

$$\{\{p_1, p_4\}, \{\neg p_1, p_3, p_4\}, \{\neg p_3, p_4\}, \{\neg p_4, p_5\}, \{\neg p_4, p_6\}, \{\neg p_5, \neg p_6\}\}$$

There are no unit clauses or pure literals, forcing us to resort to splitting, and we randomly opt to split on p_4. For the p_4 case, we call DPLL with argument:

$$\{\{p_5\}, \{p_6\}, \{\neg p_5, \neg p_6\}\}$$

Applying UNIT results in:

$$\{\{p_6\}, \{\neg p_6\}\}$$

and another application of UNIT results in $\{\{\}\}$, so that we can conclude that this side of the split is unsatisfiable.
For the $\neg p_4$ case, we call DPLL with:

$$\{\{p_1\}, \{\neg p_1, p_3\}, \{\neg p_3\}, \{\neg p_5, \neg p_6\}\}$$

Applying UNIT results in:

$$\{\{p_3\}, \{\neg p_3\}, \{\neg p_5, \neg p_6\}\}$$

and another application of UNIT (on $\{p_3\}$) results in $\{\{\}, \{\neg p_5, \neg p_6\}\}$, so that we can conclude this side of the p_4 split is also unsatisfiable (note that we did not need to resolve one clause here). Since both sides of the split are unsatisfiable, we conclude that the original wff is unsatisfiable.

In the above example, we seldom had reason to split, as UNIT and PURE were able to greatly reduce the problem size in most cases. This is not an exception, and empirical evidence indicates that a large portion of total time (*e.g.*, 80%) in DPLL is spent on the simple UNIT rule for many real problems. Indeed, the UNIT rule is all that is needed for Example 1.21!

1.13 Theoretical Properties of PC

In the previous sections, we have defined a model theoretic semantics of PC denoted using the $\models$ symbol, along with several different proof procedures for PC denoted using the $\vdash_{SC}$, $\vdash_{TAB}$, $\vdash_{RES}$ symbols. Additionally, various forms of soundness and completeness were shown for different proof procedures. Although the model- and proof-theoretic semantics end up being equivalent for all these PC proof procedures, it is still important to distinguish these two as there are logics in which it can be shown that no proof system has comparable properties.

This section discusses additional properties of PC that deal with properties of the logic and proof systems themselves rather than the wffs being proven. Such properties are generally called metalogical properties. When a statement applies to all proof procedures of this chapter, we simply use the unlabeled $\vdash$ symbol.

Deduction Theorem

The first question deals with whether proving ψ from $\phi_1, \phi_2, \ldots \phi_n$ can be done by instead proving the implication $(\phi_1 \wedge \phi_2 \wedge \ldots \wedge \phi_n) \rightarrow \psi$, using sequent calculus for now. It is simple to show from the definition of $\models$ that:

$$\phi_1, \phi_2, \ldots, \phi_n \models \psi \text{ iff } \models (\phi_1 \wedge \phi_2 \wedge \ldots \wedge \phi_n) \rightarrow \psi$$

The deduction theorem states that the same holds for $\vdash_{SC}$:

Theorem 1.14 ***(Deduction Theorem):***
$\phi_1, \phi_2, \ldots, \phi_n \vdash_{SC} \psi$ *iff* $\vdash_{SC} (\phi_1 \wedge \phi_2 \wedge \ldots \wedge \phi_n) \rightarrow \psi$

Proof:

$\Rightarrow$: Suppose $\phi_1, \phi_2, \ldots, \phi_n \vdash_{SC} \psi$. Then, multiple applications of ($\wedge$ L) and (Exc L) result in

$$\underbrace{\phi_1 \wedge \ldots \wedge \phi_n, \ \ldots, \ \phi_1 \wedge \ldots \wedge \phi_n}_{\text{n times}} \vdash_{SC} \psi$$

Multiple contractions followed by an application of ($\rightarrow$ R) produces the desired result: $\vdash_{SC} (\phi_1 \wedge \phi_2 \wedge \ldots \wedge \phi_n) \rightarrow \psi$.

$\Leftarrow$: Denote $\phi_1 \wedge \phi_2 \wedge \ldots \wedge \phi_n$ as ϕ and suppose $\vdash_{SC} \phi \rightarrow \psi$.
A weakening results in:

$$\phi \vdash_{SC} \phi \rightarrow \psi$$

On another branch of the derivation tree, a slight variation of the modus ponens argument of Example 1.15 can be used to derive:

$$\phi \rightarrow \psi, \phi \vdash_{SC} \psi$$

A cut on the above two sequents results in $\phi, \phi \vdash_{SC} \psi$, and a contraction can then be used to conclude the desired sequent $\phi \vdash_{SC} \psi$.
QED

As a metalogical theorem, the deduction theorem can be used to prove various other interesting properties of proofs. For example, an important consequence of the deduction theorem is the often-used principle of proof by contradiction, one form of which is captured by the following corollary:
Corollary: If $\Gamma, \phi, \neg\psi \vdash_{SC} \neg\phi$ then $\Gamma \vdash_{SC} \phi \rightarrow \psi$.
Proof: Suppose $\Gamma, \phi, \neg\psi \vdash_{SC} \neg\phi$. By the deduction theorem, this is equivalent to $\Gamma \vdash_{SC} (\phi \wedge \neg\psi) \rightarrow \neg\phi$. The model theoretic semantics can be used to show that this is equivalent to $\Gamma \vdash_{SC} \phi \rightarrow \psi$.
QED

While the proof of the above corollary could also have been done directly as a short sequent calculus derivation (see Exercise **??**), our goal was to show it is a trivial consequence of the deduction theorem.

While the deduction theorem above is stated for sequent calculus, we have in fact used analogs of it unwittingly in earlier examples of other proof systems. For example, in the resolution example 1.21, it could be argued that the actual goal was to prove $\models (\phi_1 \wedge \phi_2 \wedge \phi_3) \rightarrow \neg w$, which is equivalent to $\vdash (\phi_1 \wedge \phi_2 \wedge \phi_3) \rightarrow \neg w$ by the soundness and completeness theorems, and which was then converted to showing $\phi_1, \phi_2, \phi_3, \neg w \underset{RES}{\vdash} \bot$ using an analog of the deduction theorem. Similarly, the tableau rules freely move wffs across the $\vdash$ symbol. Indeed, any refutation based procedure to show $\vdash \phi$ is instead showing some version of $\neg\phi \vdash \bot$, once again freely moving wffs across the turnstile. Such laxity may be excusable for a simple logic such as PC (in an introductory chapter), but caution needs to be exercised in other logics where the deduction theorem may not hold. The deduction theorem is what gives us the flexibility to transform proofs of one form into some other hopefully easier form.

The above motivated a need for the deduction theorem from the point of view of a prover, whether human or computer. From the point of view of somebody interested in the foundation of mathematics, it also relates a proof procedure $\phi \vdash \psi$ to a theorem $\models \phi \rightarrow \psi$, which reduces to $\vdash \phi \rightarrow \psi$ after applying soundness/completeness. The deduction theorem can thus be applied to essentially generate proof rules from axioms. For example, suppose an axiom $\underset{SC}{\vdash} (\phi \wedge \psi) \rightarrow \phi$ is given. The deduction theorem allows us to conclude $\phi, \psi \underset{SC}{\vdash} \phi$, which is equivalent to a proof rule inferring ϕ from $\phi \wedge \psi$. This power needs to be carefully exercised since such manipulations can not be done in logics where the deduction theorem does not hold, as [faultily] obvious they may seem to be.

Soundness and Completeness Theorems

To review, a proof system is said to be sound if everything provable in the system is in fact valid. Conversely, a proof system is complete if it is capable of proving all valid statements. Thus, an unsound proof system would seem to be of questionable utility, though an incomplete system that works in 'most' cases may seem to have some value. Note that a proof system that claims every wff to be not valid is still sound, and similarly a complete system could claim every wff to be valid – requiring both soundness and completeness avoids such silly systems. In some logics covered later, such a proof system is not always possible; however both hold for PC as we have already shown that each of the proof systems of this chapter is sound and complete. For most logics, soundness is generally easy to prove since it only involves showing that each rule in the proof system (or step in the algorithm) is sound – *i.e.*, it follows from the model theoretic semantics. Such proofs are typically mechanical, though they may be tedious to account for all forms of wffs and proof rules. Completeness is generally harder to prove in logics more complex than PC. If a logic has some complete

proof procedure, the logic itself is informally characterized as complete.

Note that the soundness and completeness theorems are stated for only finitely many wffs (and each wff itself is of course finite). Although they also apply more generally, we do not show that here. However, the following theorem relates satisfiability of infinite and finite sets of wffs. We assume the definitions elsewhere in the chapter are extended from sequences of wffs $\phi_1, \ldots, \phi_n$ to [possibly infinite] sets of wffs Γ.

Theorem 1.15 ***(Compactness Theorem):***
An infinite set of wffs Γ is satisfiable iff every finite subset $\Gamma' \subset \Gamma$ is satisfiable.

As the proof of this for the uncountable case involves additional machinery, our proof is only for countable Γ:
Proof:
$\Rightarrow$:
This part is trivial since any model that satisfies Γ must also satisfy any subset Γ', finite or otherwise.
$\Leftarrow$:
Let Γ be $\{\phi_1, \phi_2, \phi_3, \ldots\}$, and denote the (possibly infinite) set of atoms $p_1, p_2, p_3, \ldots$. Suppose every finite subset of Γ is satisfiable. For each $n \geq 1$, let Γ_n be the (possibly empty) subset of $\{\phi_1, \ldots, \phi_n\}$ consisting at most atoms in $\{p_1, \ldots, p_n\}$. For example, Γ_3 contains whichever wffs among ϕ_1, ϕ_2, and ϕ_3 contain at most the atoms p_1, p_2, and p_3 (it is possible for Γ_3 to be null). Since Γ_n is a finite subset of Γ, each Γ_n has a model $\mathcal{M}_n$.

We will now show how to construct a model $\mathcal{M}$ of Γ from the $\mathcal{M}_n$'s, thus showing that Γ is also satisfiable. To do this, we first construct a sequence of sets N_i and S_i that will contain indices of models that meet certain properties, thus referred to as index sets:

$$N_1 = \mathbb{Z}^+$$

$$S_k = \{n \in N_k : \mathcal{M}_n \models p_k\}$$

$$N_{k+1} = \begin{cases} \text{if } S_k \text{ is finite} : \{n \in N_k \backslash S_k : n \geq k+1\} \\ \text{if } S_k \text{ is infinite} : \{n \in S_k : n \geq k+1\} \end{cases}$$

This construction ensures that N_{k+1} indexes a set of models that either all model p_k or all model $\neg p_k$ depending on whichever is infinite, making the N_i's all infinite. We claim that the model $\mathcal{M}$ such that $\mathcal{M} \models p_i$ iff S_i is infinite satisfies Γ. For consider any wff $\phi \in \Gamma$. It is in some Γ_n. Since $\mathcal{M}_n \models \Gamma_n$, we also have $\mathcal{M}_n \models \phi$. But $\mathcal{M}_n$ and $\mathcal{M}$ agree on $p_1, \ldots, p_n$ by construction; thus $\mathcal{M} \models \Gamma_n$, implying $\mathcal{M} \models \phi$.
QED

The compactness theorem can be exploited to also extend numerous problems representable in PC from the finite to infinite case. A particularly notable class of applications are in graph theory since any theorems about finite graphs can be extended to infinite graphs by appealing to the compactness theorem, provided

the problem is expressible as a PC satisfiability problem. For example, the well known four color theorem for finite planar graphs also applies to infinite planar graphs since the constraints involving graph coloring may be encoded as PC wffs.

The unsatisfiability form of the compactness theorem is often more useful in logic, and it readily follows:
Corollary: An infinite set of wffs Γ is unsatisfiable iff there is some unsatisfiable finite subset $\Gamma' \subset \Gamma$.

The corollary has direct applicability to decidable refutation-based proof procedures such as tableaux, resolution and its derivatives. Recall that completeness of these systems was proven only for finite sets of wffs (the soundness proofs applied even for infinite sets since the proofs only relied on finiteness of the derivations). The compactness theorem allows us to extend these methods to infinite sets of wffs. For suppose Γ contains infinitely many wffs $\gamma_1, \gamma_2, \ldots$ and suppose $\Gamma \models \phi$. Then, $\Gamma \cup \{\neg\phi\}$ is unsatisfiable. By the above corollary, there is some finite subset $\Gamma' \subset \Gamma$ such that $\Gamma' \cup \{\neg\phi\}$ is unsatisfiable. A proof procedure can simply attempt proofs of $\vdash \phi$ (to handle the case where $\neg\phi$ is unsatisfiable), then $\gamma_1 \vdash \phi$, then $\gamma_1, \gamma_2 \vdash \phi$, then $\gamma_1, \gamma_2, \gamma_3 \vdash \phi$, and so on. Since the proof procedure is decidable, each of these attempts terminates. This procedure will thus eventually be able to conclude $\gamma_1, \ldots, \gamma_k \vdash \phi$ for some k, showing $\Gamma \vdash \phi$.

Since many applications of PC generally involve finite sets of wffs, the compactness theorem is mainly of theoretical interest in this chapter. However, it becomes important when extending to First Order Logic in Chapter 3.

Decidability

The completeness theorem states that any valid wff is also provable. But what it does not say is that there exists a way to find such a proof. Equivalently, it does not say that there is an algorithm that will terminate on all inputs with a correct answer to the VAL problem. While metalogic theorems generally involve completeness, applied computer science is often more interested in algorithms. In particular, there are two key characteristics of interest:

Def: A logic is **decidable** if there exists a decidable proof procedure for the validity problem in the logic. If it terminates on all valid wffs but not necessarily on the invalid wffs, it is called **semi-decidable**.

When presenting the tableau and resolution proof procedures, we already showed that they terminated on all inputs, resulting in the following theorem:

Theorem 1.16 *PC is decidable.*

Our proofs were simple, since they generally exploited the property that there are a finite number of atoms/models for any PC wff, and the proof procedures reduced the size of the problem (*e.g.*, by eliminating an atom) in each step.

The above theorem talks about the validity problem, but analogous statements can also be made about the decidability of the SAT problem. Although PC SAT is decidable, our procedures for CNF SAT all had exponential complexity. This raises the question of whether a faster algorithm exists. The SAT problem for PC is in fact the canonical NP-complete problem, a class of decision problems for which no polynomial time algorithm has been found. It can be shown that if there is a polynomial time algorithm for one NP-complete problem, all such problems have polynomial time algorithms. The existence of such an algorithm is a fundamental open problem in computer science, referred to as the P=NP problem. It is especially important since many of the most interesting problems in computer science, including many in graph theory, are NP-complete. A common way of proving that a problem is NP-complete is to reduce SAT to that problem – that is to show a polynomial time transformation that solves the problem given a solution to the PC SAT problem. The existence of such reductions also suggests that an effective SAT solution can also solve a huge collection of many of the most interesting problems in computer science!

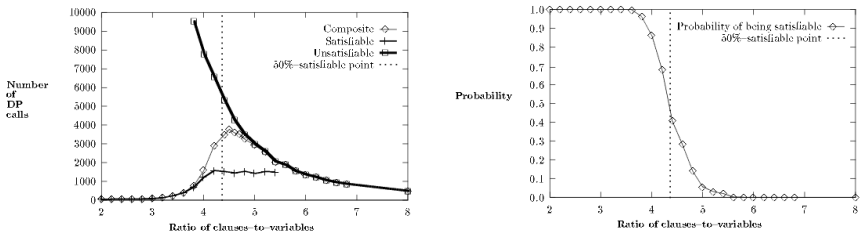
Note that the SAT problem refers to the solution of general PC wffs. There are several simpler cases of wffs that yield more efficient solutions:

- **2SAT:** This is the restriction of CNF in which each clause is limited to at most 2 literals. In this case, the SAT problem has polynomial complexity. The crucial observation exploited by most 2SAT algorithms is that a clause $\ell_1 \vee \ell_2$ is equivalent to $\neg\ell_1 \rightarrow \ell_2$, allowing the algorithm to follow the chain of implications – in contrast, a 3SAT clause $\neg\ell_1 \rightarrow (\ell_2 \vee \ell_3)$ has a choice between ℓ_2 and ℓ_3 to follow, leading to exponential blowup. The reader may also choose to ponder a specialization of resolution to this case. Unfortunately, very few applications are naturally modeled as 2SAT wffs.

- **HORNSAT:** This is similar to CNF, except that each clause must be a Horn clause (instead of an arbitrary disjunction). The forward chained algorithm discussed in Section 1.11.1 is one of several decidable polynomial complexity SAT algorithms for Horn formulas. Although several useful classes of problems can be modeled as Horn formulas, there are still problems not representable this way.

- **DNFSAT:** This is the DNF analog of the CNF SAT algorithms discussed in this chapter. It is simple to solve SAT for DNF wffs since a DNF wff is satisfiable iff one of its clauses is satisfiable, and a DNF clause is satisfiable iff it does not contain both a literal and its negation. This observation yields a simple polynomial time SAT algorithm that merely scans through the wff, looking for a DNF clause that does not contain both a literal and its negation.

Given the low complexity of DNFSAT, it is reasonable to question the use of CNF. However, a wff's DNF form may be exponentially larger than its original form, and we have thus not changed the exponential complexity of the general SAT problem. In contrast to the DNF case, it is possible to convert any wff



(a) Difficulty of random 3-SAT wffs (b) Probability of being satisfiable

Figure 1.9: Empirical Studies of DPLL on 3-SAT (figures taken from [**?**])

to an equisatisfiable CNF wff with linear growth in both number of atoms and clauses using what is called the Tseitin transformation, but the exponential time here occurs in solving SAT for the CNF form.

Although the SAT problem for PC likely has exponential complexity (assuming P $\neq$ NP), empirical studies have shown that many wffs can be solved efficiently in practice. A seminal paper in this field [**?**] explored several cases of this problem, including the one where each CNF clause has exactly three literals, a slight variation of 3SAT. They observed that the ratio of clauses to atoms is crucial to the likely efficiency of a DPLL-type procedure. Figure 1.9 illustrates the number of DPLL calls as a function of this ratio ("variables" here is short for "propositional variables", or "atoms") for a large number of randomly generated wffs over an alphabet of 50 atoms. It can be seen that the procedure is efficient for wffs with low or high clause-atom ratios, with the harder area occurring at a ratio of approximately 4.3, which also seems to correlate with the point where 50% of formulas are satisfiable. Similar experimental results where there is a relatively narrow mid-region containing many hard instances of a problem have been shown for many related problems. Of course, as with any NP-complete problem, there will always be some problems that are not practically solvable.

1.14 SAT Solvers

Despite the NP-completeness of the SAT problem, recent advances have made it solvable in practice for many cases. Looking at the DPLL algorithm, an obvious implementation strategy involves data structure to quickly identify clauses where UNIT (or PURE) applies. A more complicated optimization relies on the observation that the splitting rule represents the primary source of inefficiency since the choice of literal to split on is arbitrary. The SPLIT rule completely explores a search space based on a choice ℓ_1, and then completely explores the space based on $\neg\ell_1$. In the typical case where only one model is desired, the second search is performed only if the first search leads to a conflict. Whenever a conflict is reached, the algorithm may backtrack to the most recent split and try the alternate choice, and is thus classified as *chronological backtracking*. If

```
c This is a comment
c (p)roblem follows, with #atoms and #clauses
p cnf 3 2
c (c)lauses follow (1-based indices of literals)
1 -3 0
2 3 -1 0
```

Figure 1.10: DIMACS format for $(p_1 \wedge \neg p_3) \vee (p_2 \wedge p_3 \wedge \neg p_1)$

the subspaces are unsatisfiable due to clause conflicts not involving ℓ_1, as would often be the case, this results in redundant computation identifying the same conflict/s for each subtree. A better strategy may be to do a [possibly partial] search based on the ℓ_1 split, analyze resulting conflicts leading to unsatisfiability, and intelligently select which literal choice to backtrack to. Since this choice will typically be for a literal determined to be responsible for the conflict rather than the most recently chosen literal ℓ_1, such an approach is denoted *non-chronological backtracking*. There are many techniques for the analyses and choices going into such algorithms, and they form the basis for a class of techniques called Conflict Driven Clause Learning (CDCL), which almost all SAT solvers today are based on.

There are many different SAT solvers based on DPLL+CDCL that have achieved tremendous success, solving problems with millions of atoms in some cases. The construction of increasingly faster SAT solvers is an art that is beyond the scope of this book[11], though we outline the use of SAT solvers here.

Although there are many competing SAT solvers, almost all support a standard DIMACS CNF input format. A DIMACS CNF file contains a preamble with some header information followed by clauses, and Figure 1.10 shows an example. Any text starting with the token `c` is a comment. The preamble contains information about the number of atoms and clauses, followed by a number of clauses. In this case, it indicates that there are 3 atoms (p_1, p_2, p_3) and 2 clauses in the CNF wff. The first clause line represents that p_1 appears positively and p_3 appears negatively in the first clause, and the 0 terminates that clause (so long clauses may span multiple lines).

SAT solvers may also provide options for controlling what is output, though these do not always follow the same standards. In general, the user outputs whether or not a CNF wff is satisfiable, and if so a satisfying model. If not satisfiable, most SAT solvers today only output a message, and there are companion tools that can output a subset of clauses that are together unsatisfiable, called an **unsatisfiable core**. Often, the unsatisfiable core is minimal, though not necessarily the smallest one – *i.e.*, , it may produce an n-clause core with

[11] The interested reader is referred to the annual SAT competition hosted at `www.satcompetition.org` for more information.

no proper subset that is unsatisfiable, but there may exist another k-clause unsatisfiable core where k<n. For example, the wff

$$\overbrace{(p_1 \vee p_2) \wedge \neg p_1 \wedge \underbrace{\neg p_2}} \wedge \neg p_3 \wedge \neg p_4 \wedge \underbrace{(p_2 \vee p_3 \vee p_4)}$$

minimal unsat. core (left) and minimal unsat. core (right)

is unsatisfiable with 2 minimal unsatisfiable cores as shown. The reader should verify that every subset of either unsatisfiable core is satisfiable. A unsatisfiable core assists in understanding the source of unsatisfiability and also in debugging models, especially for large CNF wffs.

Most SAT solvers only output one model (in the satisfiable case), and this may not be the most useful model for the user needs. For the satisfiable case, it is simple to extract other models by representing a falsification of the found model in CNF, adding the corresponding clause, and rerunning. Similar additions can be made for other analyses.

Example 1.29: We revisit the 3 houses puzzle for which Example 1.6 showed a PC encoding. The problem there is clearly a SAT problem, since we need to find a satisfying model for those wffs. To use a SAT solver, we convert each wff into a CNF clause (since there is an implicit conjunction between wffs) and then represent it in DIMACS form (Figure 1.11). We then solve it using a SAT solver, producing the following output[12]:

```
SAT
1 -2 -3 -4 5 -6 -7 -8 9 -10 11 -12 13 -14 -15 -16 -17 18 0
```

Interpreting the output, a satisfying model assigns $\top$ to $p1$, $\bot$ to $p2$, and so on, resulting in the following model:

H1	blue	CS major
H2	red	MATH major
H3	white	PHIL major

In general there are multiple ways of encoding a problem, and SAT solvers may have widely varying performances depending on the encoding. Thus, it is commonplace to experiment with different encodings to achieve one with satisfactory performance on larger problems, typically aiming to reduce the number of atoms, literals, and/or clauses. Counter-intuitively, increased numbers of atoms, literals, or clauses do not always result in slower performance, and some encodings that perform well in practice actually grow the number of atoms by large amounts. Unfortunately, it is difficult for end users to predict which encodings will work for a given problem since SAT solvers embody a variety of strategies. Nevertheless, SAT solvers today are widely used by both industry and academia in numerous applications including CAD, planning in artificial

[12]The `minisat` solver is used here, and output formats vary across solvers.

```
c -------------------------------------------------------
c General constraints

p cnf 18 49

c Every house has a color
1 4 7 0    2 5 8 0    3 6 9 0

c Every house has a resident
10 13 16 0    11 14 17 0    12 15 18 0

c One house cant have two colors
-1 -4 0    -1 -7 0    -4 -7 0
-2 -4 0    -2 -8 0    -5 -8 0
-3 -6 0    -3 -9 0    -6 -9 0

c One color cant be on two houses
-1 -2 0    -1 -3 0    -2 -3 0
-4 -5 0    -4 -6 0    -5 -6 0
-7 -8 0    -7 -9 0    -8 -9 0

c One house cant have two residents
-10 -13 0    -10 -16 0    -13 -16 0
-11 -14 0    -11 -17 0    -14 -17 0
-12 -15 0    -12 -18 0    -15 -18 0

c One resident cant be in two houses
-10 -11 0    -10 -12 0    -11 -12 0
-13 -14 0    -13 -15 0    -14 -15 0
-16 -17 0    -16 -18 0    -17 -18 0

c -------------------------------------------------------
c SPECIFIC CONSTRAINTS

c The philosophy major lives directly to the right of the Red house.
-4 17 0
-5 18 0
-6 0

c The computer science major lives in the Blue house.
-1 13 0
-2 14 0
-3 15 0

c The math major lives in house two.
11 0
```

Figure 1.11: 3 Houses Puzzle SAT Problem in DIMACS Format

intelligence, formal methods, and software engineering tools. Recent years have seen an explosive (and continuing) growth in SAT solver performance, and they can now handle thousands of clauses and millions of literals. However, the nature of NP-completeness means that there will always be some problems that take too long.

There are also SAT solvers for various related problems:

MAXSAT: The MAXSAT problem is that of finding the maximum number of CNF clauses that can be satisfied by any model. There are also several variations, including:

- Partial MAXSAT: Some clauses are required to be satisfied, thus maximizing the number of satisfied remaining clauses.
- Weighted MAXSAT: Weights are associated to each clause, and the weighted sum of satisfied clauses is maximized.

A solution to MAXSAT obviously results in a solution to the SAT decision problem since all clauses are satisfied if the original CNF wff is satisfiable; thus, it also has high complexity. However, as with SAT, there are several MAXSAT solvers of practical use.

#SAT: This is the model counting problem, which determines the number of models of a SAT problem instead of a satisfying model. While #SAT may also be solved by repeated calls to a SAT solver, augmenting the clauses with the negation of a found satisfying model with each call, a #SAT checker is far more efficient.

INCSAT: The incremental SAT problem attempts to exploit commonalities between different instances of a SAT problem to make subsequent solutions faster. Suppose we wish to solve a series of SAT problems,

$$C \cup P_1, C \cup P_2, \ldots, C \cup P_n$$

where C represents clauses that are common to each of these problems and the P_i's differ. Since the conflicts in C are common to each of these problems, they can be learned to avoid wasteful searches in future instances, and such learning forms the core of INCSAT solvers. A useful special case of this problem occurs when $P_1 \subset P_2 \subset \ldots \subset P_n$, and additional conflicts involving the P_i's may also be learned in this version of the problem.

There is also an extension of SAT from PC to first order logic (to be discussed in Section 3.10). All of these problems have seen increasing applications in recent years, along with increasing tool support for efficient solutions.

Index of Symbols

Index