CS 80 Course Notes

Everett L. Bull, Jr.
Pomona College

Fall, 2000

1 Mathematical Preliminaries

These notes are a brief summary of the definitions and concepts we reviewed in class. Not all the definitions are rigorous, but they will suffice for our purposes.

1.1 Sets

We adopt an intuitive approach to sets. A set is composed of elements.

- We use the symbol $\in$ to denote element-hood or membership. The notation $a \in B$ means that a is an element, or a member, of the set B.
- The set without elements is the empty set, $\emptyset$.
- Two sets are equal if they have the same elements. Repetition and order of the elements in the set make no difference.
- Sets may be described by listing their elements or by giving a description of them. We use brackets to delimit sets.
  - $S = \{red, green, yellow\}$.
  - $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$, the set of natural numbers.
  - $\mathbb{Z} = \{\ldots, -1, 0, 1, 2, \ldots\}$, the set of integers.
  - P is the set of positive prime integers.

  One must be a bit careful in describing sets. For example, we might be tempted to add to the above list:

  - U is the "set" of all sets.

But there is no such set. (If there were, we would encounter a contradiction with the Russell Paradox.) One is safe, however, when describing a set that is a subset of a known set. For example, the set of positive prime numbers is a subset of $\mathbb{N}$.

$$P = \{n \in \mathbb{N} \mid 1 < n \text{ and the only divisors of } n \text{ are 1 and } n\}.$$

- Among the operations on sets are $\cup$, $\cap$, $\overline{}$, $\mathcal{P}(\ \ )$, and $\setminus$, denoting union, intersection, complementation, power set, and subtraction, respectively.
- The inclusion, or subset, relation is denoted by $\subset$ or $\subseteq$. These are usually used interchangeably. The symbol $\subsetneqq$ is used to denote *proper* inclusion.

1.2 Tuples

An **ordered n-tuple** is composed of n elements, with a specified order. Two n-tuples are equal if they have the same elements in the same order. Repetitions are allowed. We use angle brackets (or sometimes parentheses) to express n-tuples:

$$\langle a_1, a_2, \ldots, a_n \rangle$$

An ordered 2-tuple is a **pair,** and an ordered 3-tuple is a **triple.**

In formal set theory the ordered pair is defined as

$$\langle x, y \rangle = \{\{x\}, \{x, y\}\}.$$

This is the Banach-Kuratowski ordered pair. One can prove that $\langle x, y \rangle = \langle x', y' \rangle$ if and only if $x = x'$ and $y = y'$. One then defines the triple $\langle x, y, z \rangle$ as $\langle \langle x, y \rangle, z \rangle$, continuing analogously for longer tuples.

The **cartesian product** of sets $S_1, S_2, \ldots S_n$ is the set of all n-tuples $\langle a_1, a_2, \ldots, a_n \rangle$ in which a_i is an element of S_i. It is denoted $S_1 \times S_2 \times \ldots \times S_n$. An important special case occurs when n is 2. The cartesian product of A and B is

$$A \times B = \{\langle a, b \rangle \mid a \in A \text{ and } b \in B\}.$$

1.3 Relations

An n-ary relation on the sets $S_1, S_2, \ldots S_n$ is a subset of the product $S_1 \times S_2 \times \ldots \times S_n$.

Important special cases are the unary (1-ary) relations, called **properties,** and the **binary** (2-ary) relations. If R is a binary relation on sets A and B, we often write $a\ R\ b$ instead of $\langle a, b\rangle \in R$. When A and B are the same, we say simply that R is a relation on A.

The set $\mathbb{E}$ of even natural numbers is a unary relation on $\mathbb{N}$. In some situations, we *define* the notion of even-ness in terms of the set $\mathbb{E}$. That gives rise to a an apparent circularity, though: Which comes first, the concept or the set? We sometimes say that the concept of even-ness is the "intension," while the set $\mathbb{E}$ is the "extension." We capture the intension with the formula

$$\exists y\ \ (y + y = x)$$

where the symbol $\exists$ is read "there exists" and the variables x and y range over the natural numbers. Any number x which satisfies the formula is even, and any number x which does not satisfy the formula is not even. We capture the extension with the set or property described as follows:

$$\{x \in \mathbb{N} \mid \exists y\ \ (y + y = x)\}\,.$$

In a similar way, we can define the extension of "less than" on natural numbers by exhibiting a binary relation, $\{\langle m, n\rangle \mid m \in \mathbb{N}, n \in \mathbb{N}$, and m is less than $n\}$. One can *define* the relations $\mathbb{E}$ and $<$ without regard to the intuitive notions of even-ness or less-than. In this course, we take the existence of both the intensions and extensions as given.

If R is a binary relation on sets A and B, the **domain** of R is the set

$$\text{domain}(R) = \{a \in A \mid \text{there exists a } b \in B \text{ such that } \langle a, b\rangle \in R\}\,.$$

The **range** of the relation is the set

$$\text{range}(R) = \{b \in B \mid \text{there exists a } a \in A \text{ such that } \langle a, b\rangle \in R\}\,.$$

Suppose that we are given two binary relations, R between A and B and S between B and C. Their **composition,** denoted $R \circ S$, is a relation between A and C defined by $R \circ S = \{\langle a, c\rangle \mid \text{there is } b \in B \text{ such that } \langle a, b\rangle \in R \text{ and } \langle b, c\rangle \in S\}$.

1.4 Special Binary Relations I: Functions

A binary relation R on sets A and B is a **functional relation** if for each $a \in A$ there is *at most* one $b \in B$ such that $a\ R\ b$.

A **partial function** is a triple $f = \langle A, G, B\rangle$ in which A and B are sets and G is a functional relation on A and B. Notice that any or all of the three sets A, B, and G can be empty. The relation G is called the **graph** of f. In informal usage, we often use f and G interchangeably.

If $f = \langle A, G, B\rangle$ is a partial function and $x \in \text{domain}(G)$, we use the notation $f(x)$ to refer to the unique element y such that $x\ G\ y$.

The partial function $f = \langle A, G, B\rangle$ is a **total function** if $A = \text{domain}(G)$.

1.5 Special Binary Relations II: Equivalence Relations

Suppose that R is a binary relation on a set A.

- The relation R is **symmetric** if for all $x, y \in A$, $x\ R\ y$ implies $y\ R\ x$.
- The relation R is **reflexive** if for all $x \in A$, $x\ R\ x$.
- The relation R is **transitive** if for all $x, y, z \in A$, $x\ R\ y$ and $y\ R\ z$ implies $x\ R\ z$.
- The relation R is **anti-symmetric** if for all $x, y \in A$, $x\ R\ y$ and $y\ R\ x$ implies $x = y$.
- The relation R is **anti-reflexive** if for all $x \in A$, $x\ R\!\!\!/\ x$.
- The relation R is **anti-transitive** if for all $x, y, z \in A$, $x\ R\ y$ and $y\ R\ z$ implies $x\ R\!\!\!/\ z$.

An **equivalence relation** on A is a symmetric, reflexive, transitive relation on A.

Let R be an equivalence relation on A. For $a \in A$, let $[a] = \{b \in A \mid a\ R\ b\}$, the **equivalence class** of the element a. The set A/R of all equivalence classes is called the **quotient** of A by R.

Proposition 1.1 *If R is an equivalence class on a set A, each element $a \in A$ belongs to* exactly *one element of A/R.*

The relation $\{\langle a, [a]\rangle \mid a \in A\}$ is the graph of a total function whose range is all of A/R. Such a function is called **onto** or **surjective.**

Example 1.1 Let n be a positive integer. The binary relation $\equiv_n$ defined by

$$x \equiv_n y \text{ if and only if } x - y \text{ is a multiple of } n$$

is an equivalence relation on $\mathbb{Z}$. The quotient set $\mathbb{Z}/\equiv_n$ has n elements:

$$\mathbb{Z}_n = \mathbb{Z}/\equiv_n = \{[0], [1], [2], \ldots, [n-1]\}.$$

Example 1.2 Let A be any set, and let

$$I_A = \{(x, x) \mid x \in A\}.$$

The relation I_A is the **identity relation** on A. It is an equivalence relation for which each equivalence class has exactly one element.

The identity relation is the "finest" equivalence relation on a set A; it makes the most distinctions among elements of A. In contrast, the "coarsest" equivalence relation is the one that makes no distinctions at all, namely $A \times A$. Under that equivalence relation, everything is equivalent to everything else, and there is only one equivalence class.

1.6 Special Binary Relations III: Orderings

The binary relation R on a set A is a **partial ordering** on A if R is reflexive, transitive, and anti-symmetric. We often use a symbol like $\leq$ for a partial ordering.

The relation R on A is a **total ordering** if it is a partial ordering and, for all $x, y \in A$, either $x \ R \ y$ or $y \ R \ x$.

The pair $(A, \leq)$ is a **partially ordered set,** or **poset,** if $\leq$ is a partial ordering on the set A.

A **chain** in a binary relation R is a sequence $\langle a_1, a_2, \ldots, a_n \rangle$ such that the a_i are all distinct and $a_i \ R \ a_{i+1}$ for $1 \leq i < n$. The chain is non-trivial if $2 \leq n$. The chain is a **cycle** if it is non-trivial and also $a_n \ R \ a_1$. A binary relation is **acyclic** if it has no cycles.

The facts in the following lemma are easy to prove and are useful in proving that relations are orderings. Proofs of parts c and d appear in Assignment 1.

Lemma 1.2 *a. A binary relation R on a set A is reflexive if and only if $I_A \subset R$.*

b. The binary relation R on A is anti-reflexive if and only if $I_A \cup R = \emptyset$.

c. The binary relation R is transitive if and only if $R \circ R \subset R$.

d. The binary relation R is a partial ordering if and only if R is reflexive and transitive and has no cycles.

The relation R on A is a **strict ordering** if it is transitive, anti-reflexive, and anti-symmetric. We often use a symbol like $<$ for a strict ordering. One can, of course, have a strict, total ordering. Parts a and b of Lemma /refLemma:ReflTransEtc can be used to prove the following connection between partial orderings and strict orderings.

Proposition 1.3 *If R is a partial ordering, then $R \setminus I_A$ is a strict ordering. If R is a strict ordering, then $R \cup I_A$ is a partial ordering.*

1.7 Transitive Closure

Let A be any set and R be an acyclic relation on A. The relation R is not necessarily an ordering on A because we have not insisted that it be reflexive, transitive, or anti-symmetric. We can, however, extend R to an ordering T on A.

Let $T_0 = I_A$, and for each natural number j, inductively define $T_{j+1} = T_j \circ R$. The next two lemmas establish properties of the relations T_j that are important for proving properties of T.

Lemma 1.4 *The following are equivalent for elements a and b of A.*

1. *$a \ T_j \ b$.*
2. *There is a finite sequence $c_0, c_1, \ldots, c_j$ of elements of A for which $c_0 = a$, $c_j = b$, and for each i satisfying $0 \le i < j$, $c_i \ R \ c_{i+1}$.*

Lemma 1.5 *For natural numbers j and k, $T_j \circ T_k \subset T_{j+k}$.*

The **transitive closure** of R is the relation $T = \cup_{j=0}^{\infty} T_j$. The following theorem justifies the name.

Theorem 1.6 *The relation T is a partial ordering on A.*

PROOF: By Lemma 1.2a, the relation T is reflexive because $I_A \subset T$.

It is a corollary of Lemma 1.5 that $T \circ T \subset T$. By Lemma 1.2c, T is transitive.

To show that T is anti-symmetric, assume that there are elements a and b of A satisfying $a \ T \ b$ and $b \ T \ a$. This means that, for some j and k, we have that $a \ T_j \ b$ and $b \ T_k \ a$. By Lemma 1.4, there is a sequence $c_0, c_1, \ldots, c_j$ of elements of A which for which $c_0 = a$, $c_j = b$, and $c_i \ R \ c_{i+1}$, for $0 \le i < j$. Also, there

is another sequence $d_0, d_1, \ldots, d_k$ for which $d_0 = b$, $d_k = a$, and $d_i \ R \ d_{i+1}$ for $0 \leq i < k$. Appending the two sequences together gives what appears to be a cycle in R, but R has no cycles. The only possibility is that all the c's and d's are equal to a. In particular c_j, which we knew to be the element b, is equal to a. We have established that $a = b$, completing the proof of anti-symmetry.

Example 1.3 Let R be the successor relation on integers. That is, uRv holds if and only if $u + 1 = v$, or equivalently, if $v - u = 1$. Then T_j is the set of all pairs $\langle u, v \rangle$ such that $v - u = j$, and T is the set of all pairs $\langle u, v \rangle$ for which $v - u$ is non-negative. The transitive closure of R, therefore, is the common $\leq$ relation.

As the following theorem shows, the transitive closure is the smallest ordering extending R.

Theorem 1.7 *Let R be an acyclic relation on a set A, and let T be the transitive closure of R. If U is an ordering on A and $R \subset U$, then $T \subset U$.*

1.8 Lower and Upper Bounds

Suppose that $(A, \leq)$ is a poset and $X \subset A$. The element $b \in A$ is a **lower bound** for X if $x \in X$ implies $b \leq x$. There may be no lower bound for X. If there is one, there may be several. A lower bound for X need not be an element of X.

A lower bound for X that is an element of X is called the **least element** of X. There is at most one such element.

An element b is a **minimal element** of X if $b \in X$ and for all $x \in X$, $x \leq b$ implies $x = b$. The difference between a minimal element and a least element is that some of the elements of X may be incomparable with the minimal element. In a total ordering, the notions of least element and minimal element coincide. Minimal elements in non-total partial orders need not be unique.

An element b is the **greatest lower bound** for X if

1. b is a lower bound for X, and
2. if c is a lower bound for X, then $c \leq b$.

The greatest lower bound, if it exits, is unique. It need not be an element of X.

One can define, analogously, an **upper bound,** a **least element,** the **least upper bound,** and a **maximal element** for the set X.

1.9 Well-Founded Sets

By now, you are comfortable with definitions and proofs by induction on the natural numbers. You are also aware that induction does not apply when the domain is the set of all integers or the set of natural numbers. One can ask, "What is so special about the natural numbers?" The answer is that the natural numbers are well-founded. A well-founded set is the most general environment where we can carry out definitions and proofs by induction.

A poset $(A, \leq)$ is **well-founded** if every non-empty subset of A has a minimal element.

Consider the following condition on a subset X of A.

$$\text{For each } x \in A, \text{ if } y \in X \text{ for each } y < x, \text{ then } x \in X. \qquad (\mathcal{I})$$

A poset $(A, \leq)$ satisfies the **principle of complete induction** if a subset X of A satisfying $(\mathcal{I})$ must be all of A.

In logical notation, the principle of complete induction can be stated like this:

$$\underbrace{(\forall x \in A)\big((\forall y \in A)\,(y < x \Rightarrow y \in X) \Rightarrow x \in X\big)}_{(\mathcal{I})} \Rightarrow X = A.$$

We can rewrite it in terms of a property $\mathcal{P}$ of A.

$$\underbrace{(\forall x \in A)\big((\forall y \in A)\,(y < x \Rightarrow \mathcal{P}(y)) \Rightarrow \mathcal{P}(x)\big)}_{(\mathcal{I})} \Rightarrow (\forall z \in A)\,\mathcal{P}(z).$$

When the principle of complete induction holds, we can use it as a technique of proof. Let X be (the extension of) some property that we want to prove for all elements of A. It suffices to prove the property $(\mathcal{I})$:

> If the property holds for all elements less than x, then it holds for x.

The usual notion of mathematical induction is a special case of the principle, applied to the poset $(\mathbb{N}, \leq)$. The induction step, assuming the result for all elements less than x and proving it for x, is just the statement above. The base case, proving the result for $x = 0$, is a special case of the statement.

Another way to think about well-founded sets is through infinite descending sequences. A sequence of elements of A is just a function from $\mathbb{N}$ into A. We usually write the values of the function as a_i instead of using the more common functional

notation $a(i)$. An **infinite descending sequence** in a poset $(A, \leq)$ is a sequence $\langle a_0, a_1, a_2, \ldots \rangle$ such that $a_{i+1} < a_i$ for $i \in \mathbb{N}$. As the next theorem shows, a partially-ordered set is well-founded exactly when there are no infinitely descending sequences.

Theorem 1.8 *The following clauses are equivalent for a poset* $(A, \leq)$.

1. *There are no infinite descending sequences in* $(A, \leq)$.
2. *The poset* $(A, \leq)$ *is well-founded.*
3. *The principle of complete induction holds for* $(A, \leq)$.

PROOF: The most direct proof would have three implications: $1 \Rightarrow 2$, $2 \Rightarrow 3$, and $3 \Rightarrow 1$. Actually we prove the contrapositive of each of the three: $\neg 2 \Rightarrow \neg 1$, $\neg 3 \Rightarrow \neg 2$, and $\neg 1 \Rightarrow \neg 3$.

First, assume that the poset is not well-founded. Then there is some subset X of A which does not have a minimal element. Let a_0 be any element of X. Since a_0 is not a minimal element, there must be some element of X less than it; let a_1 be such an element. Now, a_1 is not a minimal element either, and we can find an element a_2 less than a_1. Continuing in this manner, we can construct an infinite descending sequence.

Second, assume that the poset does not satisfy the principle of complete induction. Then there is a set X, different from A, which satisfies the condition $(\mathcal{I})$. Let Y be $A \setminus X$.[1] Clearly, Y is not empty. The set Y cannot have a minimal element. If it did, then all the predecessors of the minimal element b would be in X and, by $(\mathcal{I})$, the element b would be in X instead of Y. Therefore, the poset is not well-founded.

Finally, assume that there is an infinite descending chain $\ldots a_2 < a_1 < a_0$. Let X be the set of all elements of A which *do not* appear in the sequence. (The set X may be empty.) For $x \in X$, the statement

$$(\forall y)\,[y < x \Rightarrow y \in X] \Rightarrow x \in X$$

is true because the conclusion is true. For $x \notin X$, the statement is also true, because x is one of the a_i and there is an element less than x (namely another element further along in the sequence) which makes the hypothesis false. Therefore, we have a set X which satisfies condition $(\mathcal{I})$ and is not all of A, and the poset does not satisfy the principle of complete induction.

[1] Remember that $A \setminus X$ is the set-theoretic difference, consisting of the set of elements of A which are *not* in X.

1.10 Strings

For each $n \in \mathbb{N}$, let $[n] = \{k \in \mathbb{N} \mid k < n\}$. Then $[n]$ is a set with n elements. (Notice that this definition is slightly different from the one in Josh Hodas's notes.)

If A is any set, a **string** over A is a function $s : [n] \to A$. The set A is the **alphabet.** The **length** of the string is n. If $n = 0$, the string is the **null string,** or the **empty string,** denoted ϵ_A. The ith element of the string is $s(i)$, sometimes written s_i. The string may be described as $s_0 s_1 \ldots s_{n-1}$.

A string of length 1 is described the same way as we would write an element of A. Normally this confusion is harmless; just keep in mind that strings of length 1 and alphabet letters are distinct kinds of objects.

If s and t are strings of lengths m and n, respectively, then the **concatenation** of s and t is the string $s \cdot t$ of length $m + n$ given by

$$s \cdot t(i) = \begin{cases} s(i) & \text{if } 0 \leq i < m, \text{ and} \\ t(i - m) & \text{if } m \leq i < m + n. \end{cases}$$

We sometimes drop the dot and just write st, especially when s or t is a string of length 1.

The following proposition states some natural properties of strings. You should verify that you can prove those results from the definitions given above.

Proposition 1.9 *a.* $s \cdot \epsilon_A = \epsilon_A \cdot s = s$.

b. $(s \cdot t) \cdot u = s \cdot (t \cdot u)$.

c. The length of $s \cdot t$ is the sum of the lengths of s and t.

If $s = u \cdot v$, then u is a **prefix** of s and v is a **suffix** of s. A string t is a substring of s if there are (possibly empty) strings u and v such that $s = u \cdot t \cdot v$. A prefix, suffix, or substring of s is **proper** if it is not ϵ_A or s.

Proposition 1.10 *If s is a substring of t, and if s and t have the same length, then $s = t$.*

The set of all strings over A is denoted A^*. The set of all non-empty strings is A^+.

Suppose that $<$ is a strict ordering on an alphabet A. The induced **lexicographic order** on strings over A is defined as follows:

- $s \leq t$ if s is a prefix of t, or
- $s \leq t$ if there is an integer j, less than the length of s and less than the length of t, such that $s_i = t_i$ for $0 \leq i < j$ and $s_j < t_j$.

Lexicographic order is the "dictionary order." Words in a dictionary are arranged in the lexicographic order induced on the underlying order of the alphabet letters. There is also a lexicographic order that can be induced on ordered pairs (and other tuples) from A; see Assignment 1.

1.11 Trees

A **tree domain** D is a nonempty subset of $\mathbb{N}^*$ satisfying the following conditions.

1. If $u \in D$, then every prefix of u is also in D.
2. If $u \in D$ and $u \cdot i \in D$, then $u \cdot j \in D$ for all j less than i.

A **tree** over Σ is a function $t : D \to \Sigma$, where D is a tree domain. If $u \in D$, the element $t(u)$ is the **label** at **node** u. The **root** of the tree is the node $\epsilon_{\mathbb{N}}$.

If there is a longest string in D, the **height** of the tree is the length of that longest string. If there is no longest string in D, the tree is said to have **infinite height.**

If $u \cdot i$ is a node in a tree, then $u \cdot i$ is a **child** of u, and u is the **parent** of $u \cdot i$. A **leaf** is a node with no children.

A partial ordering on a tree domain is

$$\leq = \{\langle u, v \rangle \in D \times D \mid u \text{ is a prefix of } v\}.$$

The poset $(D, \leq)$ is well-founded. If $u, v \in D$ and $u \leq v$, then v is a **descendent** of u, and u is an **ancestor** of v.

If, further, $\text{length}(u) = m$ and $\text{length}(u) = n$, then there is a **path** from u to v of length $n - m$ which is given by

$$u, uv_m, uv_m v_{m+1}, uv_m v_{m+1} v_{m+2} \ldots, uv_m v_{m+1} \ldots v_{n-1}.$$

A **branch** through the tree is a path from the root to a leaf.

Given a tree $t : D \to \Sigma$ and an element u of D, the **subtree** rooted at u is the tree t/u whose domain is $\{v \mid u \cdot v \in D\}$ and which is defined by

$$t/u(v) = t(u \cdot v).$$

2 Propositional Logic

2.1 Syntax

Consider the alphabet Σ which is the union of the following four sets:

1. $P = \{p_0, p_1, p_2, \ldots\}$ is a set of **proposition symbols.**

2. $C = C_1 \cup C_2$ is the set of **connectives,** where $C_1 = \{\neg\}$ is the set of **unary** connectives and $C_2 = \{\wedge, \vee, \Rightarrow, \Leftarrow, \equiv\}$, the set of **binary** connectives.

3. $V = \{\top, \bot\}$ is the set of **constants.**

4. $A = \{), (\}$ is the set of **auxiliary symbols.**

The **propositional language** over P is the smallest subset Prop of Σ^* satisfying the three conditions below. See the end of this section for a digression on "the smallest set."

1. $P \cup V \subset \mathsf{Prop}$;

2. if $\Phi \in \mathsf{Prop}$, then $(\neg\Phi) \in \mathsf{Prop}$; and

3. if $\Phi, \Psi \in \mathsf{Prop}$ and $\diamond \in C_2$, then $(\Phi \diamond \Psi) \in \mathsf{Prop}$.

The set $P \cup V$ is the set of **atoms**. (Strictly speaking, in the first condition the elements of $P \cup V$ are alphabet symbols, while the elements of Prop are strings. Fortunately, the confusion between symbols and strings of length one is not harmful here.) The elements of the set Prop are **formulas.**

The reason for all the parentheses in formulas is so that there is no ambiguity in reading them. Without parentheses, a string like $a \Rightarrow b \wedge c$ has two interpretations, $((a \Rightarrow b) \wedge c)$ or $(a \Rightarrow (b \wedge c))$. It is sometimes useful to think of formulas as being presented unambiguously as trees, without parentheses. Such a tree is called a **formulation tree.** Each node is labeled with a connective or an atom, and the atoms appear at the leaves. An example is in Figure 1.

To avoid writing so many parentheses, we adopt conventions that allow us to reconstruct the full formula from a string which has some of the parentheses omitted. The connectives are assigned an order of precedence, from left to right.

$$\neg \quad \wedge \quad \vee \quad \Rightarrow \quad \Leftarrow \quad \equiv$$

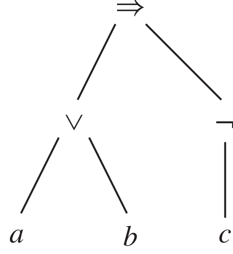

Figure 1: The formulation tree for the formula $((a \vee b) \Rightarrow (\neg c))$.

That is, $\neg$ is applied first, and then $\wedge$, and so forth. Hence, $a \Rightarrow b \wedge c$ is an abbreviation for the formula $(a \Rightarrow (b \wedge c))$. The binary connectives associate from left to right, so $a \Leftarrow b \Leftarrow c$ is an abbreviation for $((a \Leftarrow b) \Leftarrow c)$.

Digression on "smallest sets" Earlier in the section, we stated that Prop is the smallest set that satisfied three properties. We can define it more formally as follows. Let $\mathcal{Y}$ be the set of all subsets of Σ^* that satisfy the three properties on the first page.

$$\mathcal{Y} = \left\{ X \in \mathcal{P}(\Sigma^*) \mid X \text{ satisfies properties 1 through 3} \right\}$$

The set $\mathcal{Y}$ is not empty because Σ^* is in it. One can define $\mathsf{Prop}_\cap$ as

$$\mathsf{Prop}_\cap = \bigcap_{X \in \mathcal{Y}} X$$

and prove that $\mathsf{Prop}_\cap \in \mathcal{Y}$. The hardest part is keeping the "types" straight: $\mathcal{Y}$ is a set of subsets of Σ^*, and $\mathsf{Prop}_\cap$ is a subset of Σ^*. Clearly, $\mathsf{Prop}_\cap$ is the "smallest subset" of Σ^* satisfying the three properties, because it is contained in any subset of Σ^* satisfying the properties.

Alternatively, the set of propositions is the union of the sets Prop_j defined as follows:

- $\mathsf{Prop}_0 = P \cup V$.
- Prop_{j+1} is the set that includes all the elements of Prop_j together with all the formulas that can be constructed using elements of Prop_j and the connectives from C.

Let $\mathsf{Prop}_\cup = \bigcup_{j=0}^{\infty} \mathsf{Prop}_j$.

The two definitions define the same set. One can prove by induction on j that $\mathsf{Prop}_j \subseteq \mathsf{Prop}_\cap$ so that $\mathsf{Prop}_\cup \subseteq \mathsf{Prop}\cap$. Also, $\mathsf{Prop}_\cup$ satisfies the conditions to be an element of $\mathcal{Y}$, so $\mathsf{Prop}_\cup \in \mathcal{Y}$, and $\mathsf{Prop}_\cap \subseteq \mathsf{Prop}_\cup$.

2.2 Subformulas

We might be tempted to say that a formula Γ is a subformula of Δ if

1. Γ is the same formula as Δ, or
2. Δ is $(\neg\Phi)$ and Γ is a subformula of Φ, or
3. Δ is $(\Phi \diamond \Psi)$ for some $\diamond \in C_2$ and Γ is a subformula of Φ or a subformula of Ψ.

This looks like a typical definition by induction. Unfortunately, we do not yet know that we can carry out induction on the set Prop. The attempted definition is not acceptable, so we proceed by defining a relation R on Prop. The relation $\Gamma\ R\ \Delta$ holds if one of the following conditions are met.

1. $\Delta = (\neg\Gamma)$, or
2. $\Delta = (\Gamma \diamond \Psi)$ for some $\diamond \in C_2$ and some formula Ψ, or
3. $\Delta = (\Psi \diamond \Gamma)$ for some $\diamond \in C_2$ and some formula Ψ.

Notice that $\Gamma\ R\ \Delta$ implies that Γ is a substring of Δ. It follows that there can be no cycles in R. Let $\preceq$ be the transitive closure of R. If $\Gamma \preceq \Delta$, we say that Γ is a **subformula** of Δ. You should verify that this definition matches your intuition of what a subformula should be.

Theorem 2.1 *The relation* $\preceq$ *makes* $(\mathsf{Prop}, \preceq)$ *a well-founded poset.*

PROOF: By the theorem on transitive closure in the previous set of notes, $\preceq$ is a partial ordering. To show that $\preceq$ is well-founded, let X be a non-empty subset of Prop, and let Φ be a string from X of shortest length. We claim that Φ is a minimal element of X. Suppose that Ψ is another element of X and $\Psi \preceq \Phi$. Then Ψ would be an element of X which is substring of Φ. Since no string in X can be shorter than Φ, we must have $\Psi = \Phi$.

Theorem 2.1 allows us to carry out proofs by complete induction on the subformula relation.

2.3 Semantics

A function $v : P \to \{\text{True}, \text{False}\}$ is called a **valuation** or **truth assignment.** It can be extended to a function $v' : \mathsf{Prop} \to \{\text{True}, \text{False}\}$ as follows:

- $v'(\top) = \text{True}$ and $v'(\bot) = \text{False}$;
- $v'(p) = v(p)$, for each $p \in P$;
- $v'((\neg\Phi)) = \text{True}$ if $v'(\Phi) = \text{False}$, and
  $v'((\neg\Phi)) = \text{False}$ if $v'(\Phi) = \text{True}$;
- $v'((\Phi \wedge \Psi)) = \text{True}$ if $v'(\Phi) = v'(\Psi) = \text{True}$, and
  $v'((\Phi \wedge \Psi)) = \text{False}$ otherwise;
- $v'((\Phi \vee \Psi)) = \text{False}$ if $v'(\Phi) = v'(\Psi) = \text{False}$, and
  $v'((\Phi \vee \Psi)) = \text{True}$ otherwise;
- $v'((\Phi \Rightarrow \Psi)) = \text{False}$ if $v'(\Phi) = \text{True}$ and $v'(\Psi) = \text{False}$, and
  $v'((\Phi \Rightarrow \Psi)) = \text{True}$ otherwise;
- $v'((\Phi \Leftarrow \Psi)) = v'((\Psi \Rightarrow \Phi))$; and
- $v'((\Phi \equiv \Psi)) = \text{True}$ if $v'(\Phi) = v'(\Psi)$, and
  $v'((\Phi \equiv \Psi)) = \text{False}$ if $v'(\Phi) \neq v'(\Psi)$.

One can prove, by complete induction on the subformula relation, that every element of Prop is assigned a value by the clauses above. It is a bit more difficult to prove that a formula gets *only one* value. To do that, we need to know that a formula arises in only one way. This property is called **unique readability.** The key step is to show that no formula can be the proper prefix of another formula.[2] (See Assignment 2.)

It is easier to understand the extension to v' by looking at the **truth tables** for the connectives.

A	$(\neg A)$
True	False
False	True

A	B	$(A \wedge B)$	$(A \vee B)$	$(A \Rightarrow B)$	$(A \Leftarrow B)$	$(A \equiv B)$
True	True	True	True	True	True	True
True	False	False	True	False	True	False
False	True	False	True	True	False	False
False	False	False	False	True	True	True

[2] Hodas's notes speak of Prop as being "freely generated," but we avoid the term here. Proofs like the one showing that v' is well-defined are common, and even logicians get tired after repeating the same proof. Freely generated sets provide the most general framework in which to carry out these proofs. The general case is not really harder than the special case given here; it is just more difficult to understand because of the added abstraction.

2.4 Satisfiability and Validity

A valuation v is a **satisfying interpretation** for a propositional formula Φ if $v'(\Phi)$ is True. We say that the satisfying valuation v is a **model** of Φ, and we write $v \models \Phi$.

The formula Φ is **satisfiable** if there is *some* satisfying interpretation for Φ.

The formula Φ is **valid** if *all* valuations are satisfying interpretations for Φ.

A formula is **contradictory** if it is not satisfiable. It is **falsifiable** if it is not valid.

Example 2.1 The formulas $a \Rightarrow a$ and $a \vee \neg a$ are valid. The formula $a \Rightarrow b$ is satisfiable and falsifiable, while $a \wedge \neg a$ is contradictory.

Lemma 2.2 *A formula Φ is valid if and only if $(\neg\Phi)$ is contradictory.*

A set U of formulas is **(mutually) satisfiable** if there is a valuation v which is a model for each element of U. The valuation is a **model** of U.

The set U is **(mutually) contradictory** if there is no valuation which is a model for every element of U.

Example 2.2 The set $\{a \Rightarrow b, a, \neg b\}$ is contradictory, even though no proper subset is contradictory.

Lemma 2.3 *a. If U is satisfiable and $V \subset U$, then V is satisfiable.*

b. If U is satisfiable and Φ is valid, then $U \cup \{\Phi\}$ is satisfiable.

c. If U is contradictory and $U \subset V$, then V is contradictory.

d. If U is contradictory and Φ is valid, then $U \setminus \{\Phi\}$ is contradictory.

A formula Ψ is a **logical consequence** of a set U of formulas, written $U \models \Psi$, if Ψ is true in all models of U.

Lemma 2.4 *a. Every valid formula is a logical consequence of the empty set.*

b. If U is contradictory, then any *formula is a logical consequence of U. In particular, any formula is a logical consequence of $\{\bot\}$.*

c. If $U \subset V$ and $U \models \Psi$, then $V \models \Psi$.

d. If Φ is valid and $U \models \Psi$, then $U \setminus \{\Phi\} \models \Psi$.

When the set U is finite, we abuse notation and write $\Phi_1, \ldots, \Phi_n \models \Psi$ instead of the more cumbersome $\{\Phi_1, \ldots, \Phi_n\} \models \Psi$. A quick way of writing "Φ is valid" is $\models \Phi$. When applied with a single formula on the left we have $\Phi \models \Psi$. In this form, $\models$ is a relation on Prop that is reflexive and transitive but not anti-symmetric.

Theorem 2.5 $\Phi_1, \Phi_2, \ldots, \Phi_n \models \psi$ *if and only if* $\models (\Phi_1 \wedge \Phi_2 \wedge \ldots \wedge \Phi_n) \Rightarrow \Psi$.

Two formulas Φ and Ψ are **equivalent,** written $\Phi \leftrightarrow \Psi$, if any model of Φ is also a model of Ψ and *vice versa.* Another way to state the definition is to say that Φ and Ψ are equivalent if $\Phi \models \Psi$ and $\Psi \models \Phi$. By Theorem 2.5, $\Phi \leftrightarrow \Psi$ if and only if $\Phi \equiv \Psi$ is a valid formula. The relation $\leftrightarrow$ is an equivalence relation on Prop. Page 7 from Lecture 10 of Hodas's notes and Figure 2.9 of Ben-Ari's book give lists of common equivalences. Among them is

$$(\neg A \Rightarrow B \wedge \neg B) \leftrightarrow A$$

which justifies proof by contradiction. When we write equivalences, we use upper-case Roman letters to represent arbitrary formulas. The expression above is really a **schema** which represents an infinite number of equivalences of the same form.

2.5 Substitution

The notion of subset is surprisingly delicate; many incorrect definitions have been proposed over the last century. Fortunately for us right now, substitution is relatively uncomplicated for propositional logic.

We want to define the result of "substituting Δ for all occurrences the subformula Γ in the formula Φ." This is simple if Γ does not occur at all in Φ; the formula Φ remains unchanged.

It is also straightforward in other cases. For example, consider replacing every occurrence of p with $q \Rightarrow r$ in $p \Rightarrow (q \Rightarrow p)$. There are two occurrences of p. The result is $(q \Rightarrow r) \Rightarrow (q \Rightarrow (q \Rightarrow r))$. Notice that we introduce extra parentheses so that there is no ambiguity. Formally, substitution is defined on the full formulas with all the parentheses, so there is never any question about which parentheses to add.

The only possible problem arises when there are several occurrences of Γ in Φ; we must replace them all at once. If we try to do it sequentially, the process may never end, because Γ may appear as a subformula of Δ. If we replace every occurrence

of p with $p \Rightarrow p$ in $p \Rightarrow (q \Rightarrow p)$, we obtain $(p \Rightarrow p) \Rightarrow (q \Rightarrow (p \Rightarrow p))$, and the process ends there.

The notation $\Phi[\Gamma := \Delta]$ stands for the result of replacing in Φ all occurrences of Γ with Δ. It is defined by induction on subformulas:

1. If $\Phi = \Gamma$ and Φ is *any* formula, then $\Phi[\Gamma := \Delta]$ is Δ.

2. If $\Phi \neq \Gamma$ and Φ is atomic, then $\Phi[\Gamma := \Delta]$ is Φ.

3. If $\Phi \neq \Gamma$ and $\Phi = (\neg\Psi)$, then $\Phi[\Gamma := \Delta]$ is $(\neg\Psi[\Gamma := \Delta])$.

4. If $\Phi \neq \Gamma$ and $\Phi = (\Psi \diamond \Lambda)$ for some binary connective $\diamond$, then $\Phi[\Gamma := \Delta]$ is $(\Psi[\Gamma := \Delta] \diamond \Lambda[\Gamma := \Delta])$.

Note that this definition is complete; the first two cases cover all the atomic formulas. Consider the formal version of the substitution mentioned above.

$$
\begin{array}{lll}
(p \Rightarrow (q \Rightarrow p))[p := p \Rightarrow p] & & \\
\quad = p[p := p \Rightarrow p] \Rightarrow (q \Rightarrow p)[p := p \Rightarrow p] & & \text{by clause 4} \\
\quad = (p \Rightarrow p) \Rightarrow (q \Rightarrow p)[p := p \Rightarrow p] & & \text{by clause 1} \\
\quad = (p \Rightarrow p) \Rightarrow (q[p := p \Rightarrow p] \Rightarrow p[p := p \Rightarrow p]) & & \text{by clause 4} \\
\quad = (p \Rightarrow p) \Rightarrow (q \Rightarrow p[p := p \Rightarrow p]) & & \text{by clause 2} \\
\quad = (p \Rightarrow p) \Rightarrow (q \Rightarrow (p \Rightarrow p)) & & \text{by clause 1}
\end{array}
$$

Lemma 2.6 *If Γ does not appear as a subformula of Φ, then $\Phi[\Gamma := \Delta]$ is Φ.*

Theorem 2.7 (Substitution Theorem) *If $\Gamma \leftrightarrow \Delta$, then $\Phi \leftrightarrow \Phi[\Gamma := \Delta]$.*

Both the lemma and theorem are proved by induction on the subformula relation. They are good places to practice writing proofs by induction.

Here are a few common equivalent forms, which we will use below in a sample calculation using the Substitution Theorem.

$$
\begin{array}{ll}
A \Rightarrow B & \leftrightarrow \neg A \vee B \\
A \vee B & \leftrightarrow B \vee A \\
A \vee (B \vee C) & \leftrightarrow (A \vee B) \vee C \\
\neg A \vee A & \leftrightarrow \top \\
\top \vee A & \leftrightarrow \top
\end{array}
$$

See if you can determine which substitution is being done at each step.

$$\begin{aligned}
U \Rightarrow (V \Rightarrow U) &\leftrightarrow \neg U \vee (V \Rightarrow U) \\
&\leftrightarrow \neg U \vee (\neg V \vee U) \\
&\leftrightarrow \neg U \vee (U \vee \neg V) \\
&\leftrightarrow (\neg U \vee U) \vee \neg V \\
&\leftrightarrow \top \vee \neg V \\
&\leftrightarrow \top
\end{aligned}$$

2.6 Complete Sets of Connectives

Among the equivalences are

$$\begin{aligned}
A \Leftarrow B &\leftrightarrow B \Rightarrow A \\
A \equiv B &\leftrightarrow (A \Rightarrow B) \wedge (B \Rightarrow A) \\
A \Rightarrow B &\leftrightarrow \neg A \vee B \\
A \vee B &\leftrightarrow \neg(\neg A \wedge \neg B)
\end{aligned}$$

Starting with a formula Φ, one can make substitutions to eliminate all instances of the connectives $\Leftarrow$, $\equiv$, $\Rightarrow$, and $\vee$ and obtain an equivalent formula that contains only the connectives $\wedge$ and $\neg$. We say that the set $\{\wedge, \neg\}$ is **complete**, because any formula is equivalent to one using only the connectives in the set.

The sets $\{\vee, \neg\}$ and $\{\Rightarrow, \neg\}$ are also complete.

There are two logical operations, which do not appear as connectives in our language, which are complete by themselves. They are $\downarrow$ and $\uparrow$, which are sometimes called **nor** and **nand**. They satisfy the following equivalences.

$$\begin{aligned}
A \downarrow B &\leftrightarrow \neg(A \vee B) \\
A \uparrow B &\leftrightarrow \neg(A \wedge B)
\end{aligned}$$

One can see that $\neg A \leftrightarrow (A \downarrow A)$, and $A \wedge B \leftrightarrow (\neg A \downarrow \neg B)$, so all the other connectives can be defined in terms of $\downarrow$. Similarly, $\uparrow$ is by itself complete. These are the only two connectives with that property. The existence of small sets of complete connectives is important in analyzing digital circuits.

2.7 Hilbert Systems

The idea in any system of proof is to start with some **axioms** and construct proofs of other formulas using some **rules of inference.** A formula that has a proof is

called a **theorem.** For the present, we seek systems that will prove all the valid formulas.

There are several systems of proof which yield all of the valid formulas of propositional logic. One such system is the Hilbert system, which (among other things) is used to study mathematical deductions.

Notice the connection between the syntactic notion of *proof* and the semantic notion of *truth.* A system is **sound** if every theorem is valid. A system is **complete** if every valid formula is a theorem.

The Hilbert system begins with the connectives $\neg$ and $\Rightarrow$. All of the other connectives are considered to be abbreviations.

$A \vee B$ means $\neg A \Rightarrow B$
$A \wedge B$ means $\neg(A \Rightarrow \neg B)$
$A \Leftarrow B$ means $B \Rightarrow A$
$A \equiv B$ means $(A \Rightarrow B) \wedge (B \Rightarrow A)$

There are three axiom schema: Each scheme represents an infinite number of axioms of the same form.

1. $A \Rightarrow (B \Rightarrow A)$
2. $(A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C))$
3. $(\neg B \Rightarrow \neg A) \Rightarrow (A \Rightarrow B)$

There is one rule of inference, *modus ponens*. It says that from A and $A \Rightarrow B$ we may deduce B. We write rules of inference using a horizontal bar, with the premises above the bar and the conclusion below.

$$\frac{A \quad A \Rightarrow B}{B} \qquad \textit{Modus ponens}$$

A **deduction** or **proof** in the Hilbert system is a sequence $\Phi_1, \Phi_2, \ldots, \Phi_n$ of formulas, each of which is either an instance of an axiom or follows from earlier formulas by *modus ponens*. The last formula Φ_n is the **theorem**. Here is a sample deduction of the theorem $A \Rightarrow A$.

1.	$A \Rightarrow (B \Rightarrow A)$	Axiom 1
2.	$A \Rightarrow ((B \Rightarrow A) \Rightarrow A)$	Axiom 1
3.	$(A \Rightarrow ((B \Rightarrow A) \Rightarrow A)) \Rightarrow$ $((A \Rightarrow (B \Rightarrow A)) \Rightarrow (A \Rightarrow A))$	Axiom 2
4.	$(A \Rightarrow (B \Rightarrow A)) \Rightarrow (A \Rightarrow A)$	2, 3, and *modus ponens*
5.	$A \Rightarrow A$	1, 4, and *modus ponens*

Example 2.3 Deductions using only *modus ponens* are intricate and difficult to find. We introduce **derived rules of inference** to make it easier to construct proofs. A derived rule of inference is written

$$\frac{A_1, A_2, \ldots A_n}{B} \qquad \text{Derived Rule}$$

meaning that if we have proofs of the hypotheses A_1 through A_n, then we also have a proof of B. We justify a derived rule of inference by showing how to construct the proof of B from the proofs of the hypotheses. One example of a derived rule is the transitivity rule, which you will verify in Assignment 3.

$$\frac{A \Rightarrow B \quad B \Rightarrow C}{A \Rightarrow C} \qquad \text{Transitivity Rule}$$

Example 2.4 Another derived rule of inference is the exchange of hypotheses.

$$\frac{A \Rightarrow (B \Rightarrow C)}{B \Rightarrow (A \Rightarrow C)} \qquad \text{Exchange of Hypotheses}$$

If one has a k-step proof of $A \Rightarrow (B \Rightarrow C)$, then it can be extended to a proof of $B \Rightarrow (A \Rightarrow C)$, as shown below.

k.	$A \Rightarrow (B \Rightarrow C)$	
$k+1$.	$(A \Rightarrow (B \Rightarrow C)) \Rightarrow$ $((A \Rightarrow B) \Rightarrow (A \Rightarrow C))$	Axiom 2
$k+2$.	$(A \Rightarrow B) \Rightarrow (A \Rightarrow C)$	k, $k+1$, and *modus ponens*
$k+3$.	$B \Rightarrow (A \Rightarrow B)$	Axiom 1
$k+4$.	$B \Rightarrow (A \Rightarrow C)$	$k+2$, $k+3$, and Transitivity

Here is an example of a proof that uses Transitivity and Exchange of Hypothesis. The theorem is $(\neg A \Rightarrow A) \Rightarrow A$, which can be considered as justifying one form of proof by contradiction. ("Assume that A is false and prove that A is true, thereby deriving a contradiction. Conclude that the original assumption must have been wrong, and therefore A is true.")

Theorem 2.8 *The formula* $(\neg A \Rightarrow A) \Rightarrow A$ *is a theorem in the Hilbert system.*

PROOF: Let $\mathcal{T}$ be any axiom.

1.	$\mathcal{T}$	Axiom
2.	$\neg A \Rightarrow (\neg\neg\mathcal{T} \Rightarrow \neg A)$	Axiom 1
3.	$(\neg\neg\mathcal{T} \Rightarrow \neg A) \Rightarrow (A \Rightarrow \neg\mathcal{T})$	Axiom 3
4.	$\neg A \Rightarrow (A \Rightarrow \neg\mathcal{T})$	2, 3, and Transitivity
5.	$(\neg A \Rightarrow (A \Rightarrow \neg\mathcal{T})) \Rightarrow$ $((\neg A \Rightarrow A) \Rightarrow (\neg A \Rightarrow \neg\mathcal{T}))$	Axiom 2
6.	$(\neg A \Rightarrow A) \Rightarrow (\neg A \Rightarrow \neg\mathcal{T})$	4, 5, and *modus ponens*
7.	$(\neg A \Rightarrow \neg\mathcal{T}) \Rightarrow (\mathcal{T} \Rightarrow A)$	Axiom 3
8.	$(\neg A \Rightarrow A) \Rightarrow (\mathcal{T} \Rightarrow A)$	6, 7, and Transitivity
9.	$\mathcal{T} \Rightarrow ((\neg A \Rightarrow A) \Rightarrow A)$	8 and Exchange of Hypothesis
10.	$(\neg A \Rightarrow A) \Rightarrow A$	1, 9, and *modus ponens*

The use of the irrelevant axiom $\mathcal{T}$ is strikingly odd. Can you find a proof that does not use it?

Example 2.5 Still another derived rule of inference is another form of proof by contradiction.

$$\frac{\neg A \Rightarrow B \quad \neg A \Rightarrow \neg B}{A} \qquad \text{Proof by Contradiction}$$

We give an informal justification. See if you can reconstruct the formal version. Using Axiom 3 and the right-hand premise, obtain $B \Rightarrow A$. Then, by Transitivity with the left-hand premise, get $\neg A \Rightarrow A$. Finally, invoke Theorem 2.8 to conclude A.

Example 2.6 The elimination of double negation is an often-used derived rule of inference.

$$\frac{\neg\neg A}{A} \qquad \text{Double Negation}$$

Suppose that we have a k-step derivation of $\neg\neg A$. We extend the derivation as follows:

k.	$\neg\neg A$	
$k+1$.	$\neg\neg A \Rightarrow (\neg A \Rightarrow \neg\neg A)$	Axiom 1
$k+2$.	$\neg A \Rightarrow \neg\neg A$	k, $k+1$, and *modus ponens*
$k+3$.	$(\neg A \Rightarrow \neg\neg A) \Rightarrow (\neg A \Rightarrow A)$	Axiom 3
$k+4$.	$\neg A \Rightarrow A$	$k+2$, $k+3$, and *modus ponens*
$k+5$.	$(\neg A \Rightarrow A) \Rightarrow A$	Theorem 2.8
$k+6$.	A	$k+4$, $k+5$, and *modus ponens*

Notice that the extension could be adapted to a direct proof of $\neg\neg A \Rightarrow A$. There is very little *operational* difference between Double Negation as a derived rule and

the use of the theorem $\neg\neg A \Rightarrow A$. The Deduction Theorem, below, makes the connection clear.

We extend the bar notation for rules of inference to cover proofs with hypotheses. The expression $\dfrac{A_1, A_2, \ldots A_n}{B}$ means that there is a proof of B in which each line is either an axiom or one of the A_i's, or follows from preceding lines by *modus ponens*.

One really important result is called the Deduction Rule, or Deduction Theorem. It justifies our usual method of proving the implication $A \Rightarrow B$, namely "assume A and prove B."

Theorem 2.9 (Deduction Theorem) *If* $\dfrac{D_1, D_2, \ldots D_k, A}{B}$ *then* $\dfrac{D_1, D_2, \ldots D_k}{A \Rightarrow B}$.

PROOF: Suppose that $\Phi_1, \Phi_2, \ldots, \Phi_n$ is a derivation of B from $D_1, D_2, \ldots, D_k$, and A. We construct a longer derivation in which the formulas $A \Rightarrow \Phi_1$, $A \Rightarrow \Phi_2$, $\ldots$, $A \Rightarrow \Phi_n$ appear, in order.

The construction is by induction on n. Suppose that we have a derivation in which $A \Rightarrow \Phi_1$, $A \Rightarrow \Phi_2$, $\ldots$, $A \Rightarrow \Phi_{n-1}$ appear in order. There are four cases.

1. Φ_n is an instance of an axiom. Then add the following lines to the new derivation.

Φ_n	Axiom
$\Phi_n \Rightarrow (A \Rightarrow \Phi_n)$	Axiom 1
$A \Rightarrow \Phi_n$	*modus ponens*

2. Φ_n is A. Then add to the new derivation the five lines of the derivation (above) of $A \Rightarrow A$.

3. Φ_n is D_i. As in the case of an axiom, add the following lines to the new derivation.

D_i	Hypothesis
$D_i \Rightarrow (A \Rightarrow D_i)$	Axiom 1
$A \Rightarrow D_i$	*modus ponens*

4. Φ_n in the original derivation is the result of applying *modus ponens* to earlier formulas, say Φ_m and $\Phi_m \Rightarrow \Phi_n$. Then the new derivation contains $A \Rightarrow \Phi_m$ and $A \Rightarrow (\Phi_m \Rightarrow \Phi_n)$. Add the following lines to the new derivation.

$(A \Rightarrow \Phi_m) \Rightarrow$ $((A \Rightarrow (\Phi_m \Rightarrow \Phi_n)) \Rightarrow (A \Rightarrow \Phi_n))$	Axiom 2
$(A \Rightarrow (\Phi_m \Rightarrow \Phi_n)) \Rightarrow (A \Rightarrow \Phi_n)$	*modus ponens*
$A \Rightarrow \Phi_n$	*modus ponens*

Notice that the D_i may appear as hypotheses in the new demonstration, but A does not. The longest addition, in case 2, is five lines, so the new demonstration is at most five times as long as the original one.

This completes the proof of the Deduction Theorem. Notice that Axioms 1 and 2 were chosen expressly for the proof of the Deduction Theorem.

Table 1 presents a list of commonly-used (derived) rules of inference, many of which correspond to standard mathematical arguments. You may want to see if you can justify them.

2.8 Soundness and Completeness

We listed many of the rules of inference to show that ordinary mathematical argument forms can be expressed in the Hilbert system. Another goal is to develop a proof system which proves all—and only—the valid formulas. Recall that a system is **sound** if all its theorems are valid, and it is **complete** if all valid formulas are theorems. The Hilbert system is both sound and complete.

Theorem 2.10 (Soundness Theorem) *If a formula Φ is provable in the Hilbert system, then Φ is valid.*

Theorem 2.11 (Completeness Theorem) *If a formula Φ is valid, then Φ is provable in the Hilbert system.*

The proof of soundness is easy. One just verifies that all the axioms are valid and that *modus ponens* preserves validity. The proof is usually cast as an induction on the length of a demonstration, showing that every demonstration ends in a valid formula.

Completeness says that every valid formula has a proof. One strategy to prove completeness is to start with a formula and try to find a proof. Under ideal conditions, one would either find a proof or discover a valuation that made the formula

Rule	Schema
Modus ponens	$\dfrac{A \quad A \Rightarrow B}{B}$
Deduction Theorem	If $\dfrac{D_1, \ldots, D_n, A}{B}$, then $\dfrac{D_1, \ldots, D_n}{A \Rightarrow B}$.
Contrapositive Rule	$\dfrac{\neg B \Rightarrow \neg A}{A \Rightarrow B}$
Transitivity Rule	$\dfrac{A \Rightarrow B \quad B \Rightarrow C}{A \Rightarrow C}$
Exchange of Hypothesis	$\dfrac{A \Rightarrow (B \Rightarrow C)}{B \Rightarrow (A \Rightarrow C)}$
Double Negation	$\dfrac{\neg\neg A}{A}$
Inverse Contrapositive Rule	$\dfrac{A \Rightarrow B}{\neg B \Rightarrow \neg A}$
Inverse Double Negation	$\dfrac{A}{\neg\neg A}$
And-Introduction	$\dfrac{A \quad B}{A \wedge B}$
Or-Introduction	$\dfrac{B}{A \vee B}$
And-Elimination	$\dfrac{A \wedge B}{A} \qquad \dfrac{A \wedge B}{B}$
Commutativity Rules	$\dfrac{A \wedge B}{B \wedge A} \qquad \dfrac{A \vee B}{B \vee A}$
Associativity Rules	$\dfrac{A \wedge (B \wedge C)}{(A \wedge B) \wedge C} \qquad \dfrac{(A \wedge B) \wedge C}{A \wedge (B \wedge C)}$
	$\dfrac{A \vee (B \vee C)}{(A \vee B) \vee C} \qquad \dfrac{(A \vee B) \vee C}{A \vee (B \vee C)}$
Distributivity Rules	$\dfrac{A \wedge (B \vee C)}{(A \wedge B) \vee (A \wedge C)} \qquad \dfrac{A \vee (B \wedge C)}{(A \vee B) \wedge (A \vee C)}$
DeMorgan Laws	$\dfrac{\neg(A \vee B)}{\neg A \wedge \neg B} \qquad \dfrac{\neg(A \wedge B)}{\neg A \vee \neg B}$
Law of the Excluded Middle	$\dfrac{}{A \vee \neg A}$
Law of the Excluded Miracle	$\dfrac{}{\neg(A \wedge \neg A)}$
Proof by cases	$\dfrac{A \Rightarrow B \quad \neg A \Rightarrow B}{B}$
Proof by contradiction	$\dfrac{\neg A \Rightarrow B \quad \neg A \Rightarrow \neg B}{A}$

Table 1: Common derived rules of inference in the Hilbert system.

false. That would give us a proof of the assertion, "If Φ is not provable, then there is some valuation that makes Φ false." Equivalently it is, "If Φ is not provable, then Φ is not valid," which is the contrapositive of the completeness property.

The difficulty with this approach is that there is no apparent strategy for searching through all possible proofs. As we have seen, proofs in the Hilbert system are not easy to motivate, and one cannot systematically "find" a proof. We do not attempt to prove a completeness theorem for the Hilbert system here. Instead, we investigate other proof systems in which the proofs are more "natural" and more amenable to algorithmic construction.

3 Natural Deduction and Sequent Calculi

In this set of notes, we investigate some alternatives to the Hilbert system of proof for the propositional calculus.

3.1 The Natural Deduction System

Gentzen's system of Natural Deduction has several rules of inference but no axioms. The rules are listed in Table 2, and most of them are self explanatory.

There is only one notational convention that we have not yet discussed. If we assume Ψ and deduce Φ we could write out the steps.

$$\begin{array}{c}\Psi\\ \vdots\\ \Phi\end{array}$$

The rule $\Rightarrow_I$ is equivalent to the Deduction Theorem for the Hilbert system; it tells us that we can then infer $\Psi \Rightarrow \Phi$.

$$\dfrac{\begin{array}{c}\not\Psi\\ \vdots\\ \Phi\end{array}}{\Psi \Rightarrow \Phi}\Rightarrow_I$$

During the proof of Φ, the formula Ψ is an unproved assumption. When we finally conclude $\Psi \Rightarrow \Phi$, the formula Ψ is no longer an assumption, and it is shown crossed out in the rule. We call that process "discharging the assumption."

Introduction Rules	Elimination Rules
$\dfrac{\begin{array}{c}\not\Psi \\ \vdots \\ \Phi\end{array}}{\Psi \Rightarrow \Phi} \Rightarrow_I$	$\dfrac{\Psi \quad \Psi \Rightarrow \Phi}{\Phi} \Rightarrow_E$
$\dfrac{\Phi \quad \Psi}{\Phi \wedge \Psi} \wedge_I$	$\dfrac{\Phi \wedge \Psi}{\Phi} \wedge_E \qquad \dfrac{\Phi \wedge \Psi}{\Psi} \wedge_E$
$\dfrac{\Phi}{\Phi \vee \Psi} \vee_I \qquad \dfrac{\Psi}{\Phi \vee \Psi} \vee_I$	$\dfrac{\Phi \vee \Psi \quad \begin{array}{c}\not\Phi \\ \vdots \\ \Lambda\end{array} \quad \begin{array}{c}\not\Psi \\ \vdots \\ \Lambda\end{array}}{\Lambda} \vee_E$
$\dfrac{\begin{array}{c}\not\Psi \\ \vdots \\ \Phi\end{array} \quad \begin{array}{c}\not\Phi \\ \vdots \\ \Psi\end{array}}{\Psi \equiv \Phi} \equiv_I$	$\dfrac{\Psi \quad \Psi \equiv \Phi}{\Phi} \equiv_E \qquad \dfrac{\Phi \quad \Psi \equiv \Phi}{\Psi} \equiv_E$
$\dfrac{\begin{array}{c}\not\Phi \\ \vdots \\ \bot\end{array}}{\neg\Phi} \neg_I$	$\dfrac{\Phi \quad \neg\Phi}{\bot} \neg_E$
	$\dfrac{\bot}{\Phi} \bot_E$
	$\dfrac{\neg\neg\Phi}{\Phi} \neg\neg_E$

Table 2: Rules of inference in the Natural Deduction system of proof. The $\neg\neg_E$ rule is present only in the classical system, not in the intuitionistic system.

Example 3.1 The simplest example of a proof in the Natural Deduction system is the proof of $A \Rightarrow A$. The formula A is a one-line proof of itself, so we certainly can deduce A from A. Then we invoke the rule $\Rightarrow_I$.

$$\dfrac{\begin{array}{c}\cancel{A}\\ A\end{array}}{A \Rightarrow A}\Rightarrow_I$$

Example 3.2 One of our first exercises with the Hilbert system was to prove the transitivity of $\Rightarrow$. Here is a proof of that fact in the Natural Deduction system.

$$\dfrac{\dfrac{\dfrac{\cancel{A} \quad A \Rightarrow B}{B}\Rightarrow_E \qquad B \Rightarrow C}{C}\Rightarrow_E}{A \Rightarrow C}\Rightarrow_I$$

After the hypothesis A is discharged, we have a derivation of $A \Rightarrow C$ from the two remaining hypotheses, $A \Rightarrow B$ and $B \Rightarrow C$.

Notice that the notation is a bit ambiguous. It is not clear from the formula above exactly *when* in the deduction that the hypothesis A is discharged. The discharge in this case happens, of course, in the last step.

Example 3.3 Another simple deduction proves the theorem $(A \vee B) \Rightarrow (B \vee A)$.

$$\dfrac{\dfrac{\cancel{A \vee B} \quad \dfrac{\cancel{A}}{B \vee A}\vee_I \quad \dfrac{\cancel{B}}{B \vee A}\vee_I}{B \vee A}\vee_E}{(A \vee B) \Rightarrow (B \vee A)}\Rightarrow_I$$

The discharge of the hypotheses A and B occurs when the $\vee_E$ rule is applied, and the hypothesis $A \vee B$ is discharged with the $\Rightarrow_I$ rule. It is usually easy to sort out how to attribute the discharges, especially in proofs this short.

Example 3.4 Here is a deduction showing that B follows from $A \vee B$ and $\neg A$.

$$\dfrac{A \vee B \qquad \dfrac{\dfrac{\neg A \quad \cancel{A}}{\bot}\neg_E}{B}\bot_E \qquad \cancel{B}}{B}\vee_E$$

The hypotheses A and B have been discharged—by the $\vee_E$ rule—and the only remaining hypotheses are the desired ones, $A \vee B$ and $\neg A$.

Example 3.5 The DeMorgan laws are frequently used to "push" negation through conjunction or disjunction. We present here a proof of one direction of one of the laws: $\dfrac{\neg(\Phi \vee \Psi)}{\neg\Phi \wedge \neg\Psi}$. We first show that $\dfrac{\neg(\Phi \vee \Psi)}{\neg\Phi}$.

$$\dfrac{\dfrac{\dfrac{\not\Phi}{\Phi \vee \Psi}\vee_I \qquad \neg(\Phi \vee \Psi)}{\bot}\neg_E}{\neg\Phi}\neg_I$$

One can similarly show that $\dfrac{\neg(\Phi \vee \Psi)}{\neg\Psi}$. From these two, we obtain

$$\dfrac{\dfrac{\neg(\Phi \vee \Psi)}{\neg\Phi \quad \neg\Psi}\text{above}}{\neg\Phi \wedge \neg\Psi}\wedge_I$$

An often-effective strategy for constructing Natural Deduction proofs is to start at the bottom. In the case of the DeMorgan law, the obvious way (but not the *only* way) to conclude $\neg\Phi \wedge \neg\Psi$ is to use $\wedge_I$. The next step is to obtain $\neg\Phi$ and $\neg\Psi$ independently. Even though one can *construct* a proof from the bottom up, it is clearer to *present* it deductively, from the top down.

3.2 Intuitionist Proofs

The mathematics that you have learned so far is **classical** mathematics. It is concerned with the *truth* of propositions according to the notion of truth on which we base our semantics. A competing philosophy of mathematics, called **intuitionism,** is concerned with the more stringent criterion of *provability.* An intuitionist does not accept all the proofs that a classical mathematician would.

For example, in order to conclude that an equation has a solution, the intuitionist must construct an explicit solution. It is not enough to simply prove that a solution exists. In Assignment 3, you saw an argument that there are irrational numbers a and b for which a^b is rational. There were two possible assignments to a and b, and the argument established that *one* of the assignments was an pair of irrational numbers whose power was rational. The argument is not intuitionistically correct, because it did not tell us *which* of the assignments was the correct one.

Intuitionist logic arises frequently in computer science. One reason is that computation is inherently "constructive." We do not want to know simply that some fact

appears in a database; we want to know what the fact is. Further, just because a computation does not yield a particular result is not a *guarantee* that the result is false.

In the propositional logic, the law of the excluded middle is the assertion that a statement is either false or true. For any formula Φ, the formula $\Phi \vee \neg\Phi$ is valid. But it is not an intuitionistic theorem, because for an arbitrary Φ, we have no way of identifying which of the two disjuncts is true.

Pierce's formula, $((p \Rightarrow q) \Rightarrow p) \Rightarrow p$, is another an example of a classical, but not intuitionistic, theorem. Because Pierce's formula is difficult to justify in "common sense terms," it is sometimes cited as evidence for the intuitionist's point of view. You will encounter Pierce's formula in the assignments.

One of the advantages of the Natural Deduction system is that it is easy to differentiate between classical and intuionistic logic. As presented in Table 2, the rule for the elimination of double negation is present in the classical system but not in the intuitionistic system. From the intuitionist's point of view, a proof of $\neg\neg\Phi$ establishes only that $\neg\Phi$ is false. Something stronger is required to claim that Φ is true.

Since the classical system has all the rules of the intuitionistic system, any intuitionistic proof is also a classical proof. All of the proofs we have seen so far do not invoke the $\neg\neg_E$ rule, so they are valid in both the intuitionistic and classical systems.

Example 3.6 The "rule" for the *introduction* if the double negation is a theorem in the intuitionistic system (and hence in the classical system).

$$\dfrac{\dfrac{\Phi \quad \not\neg\Phi}{\bot}\neg_E}{\neg\neg\Phi}\neg_I$$

The classical $\neg\neg_E$ rule is equivalent to several other, familiar rules of classical logic. One is the law of the excluded middle, which in Natural Deduction could be written as an inference:

$$\dfrac{}{\Phi \vee \neg\Phi}\text{Excluded Middle}$$

Another equivalent to the $\neg\neg_E$ rule is the rule *Reductio ad absurdum.*

$$\dfrac{\begin{array}{c}\cancel{\neg\Phi}\\ \vdots \\ \bot\end{array}}{\Phi}\text{RAA}$$

Once again, we see that the intuitionistic system allows one kind of deduction, $\neg_I$, but not the seemingly similar RAA rule. How are they different?

Example 3.7 Although the law of the excluded middle is not a theorem of the intuitionistic system, its double negation is.

$$\dfrac{\dfrac{\dfrac{\dfrac{\cancel{\neg(\Phi \vee \neg\Phi)}}{\neg\Phi \wedge \neg\neg\Phi}\text{DeMorgan, above}}{\neg\Phi \qquad\qquad \neg\neg\Phi}\wedge_E}{\bot}\neg_E}{\neg\neg(\Phi \vee \neg\Phi)}\neg_I$$

In the classical system, we can extend the proof one step further by applying the $\neg\neg_E$ rule to conclude the law of the excluded middle.

3.3 Connections with the Hilbert System

All of the rules of the Natural Deduction system are derived rules in the Hilbert system. Therefore, any Natural Deduction proof is also a Hilbert proof.

Conversely, *modus ponens* is just the Natural Deduction rule $\Rightarrow_E$. If one could prove the three Hilbert axiom schema are theorems of Natural Deduction, then one would have verified that any Hilbert proof is also a Natural Deduction proof. The proof that the axioms are Natural Deduction theorems is left as an exercise.[3]

The conclusion is that the Natural Deduction system has exactly the same theorems as the Hilbert system. Any result (like soundness, completeness, or decidability) for one system will also apply to the other.

In class, we sketched a proof of the Soundness Theorem for the Hilbert System, but we deferred a proof of the Completeness Theorem. The Natural Deduction system is still not an easy context for a proof of the Completeness Theorem, but it is one step closer.

[3] The Hilbert system is a classical one, so the proof of at least one of the Hilbert axioms will require the rule $\neg\neg_E$. Which one?

3.4 Sequent Calculus

A sequent calculus is yet another way to deduce formulas. A **sequent** is an expression of the form

$$\Gamma \rightarrow \Delta$$

in which Γ and Δ are finite sequences of formulas. Only the formulas, and not the order in which they are listed, in Γ and Δ matter. The sequent can be read as follows:

> If all the formulas in Γ are true, then one (or more) of the formulas in Δ are true.

Notice that the sentence says "true," not "valid." A more precise rendering would be this:

> For each valuation v, if v makes all the formulas in Γ true, then v makes *some* formula in Δ true.

The sequent carries along both premises (to the left of the arrow) and conclusions (to the right). We can derive one sequent from another according to the rules like those Table 3. A formula Φ is a theorem if there is some sequence of applications of the rules that ends in $\rightarrow \Phi$. One can verify (do so!) that each rule is sound according to the above interpretation.

The rules in Table 3 are presented in the same style as rules of inference and can be read top-to-bottom or bottom-to-top. The first reading, from top to bottom, corresponds to the construction of a deduction, starting with the premises. The second reading, from bottom to top, is useful when searching for a proof of a particular formula.

The system in Table 3 is the **intuitionistic sequent calculus.** It is sometimes called LJ. All the rules have the property that there is only one formula on the right side of each sequent.

Example 3.8 Our standard first theorem is $A \Rightarrow A$. It is easy in the intuitionistic sequent calculus.

$$\dfrac{\dfrac{}{A \rightarrow A}\,\text{id}}{\rightarrow A \Rightarrow A}\Rightarrow_R$$

Example 3.9 The transitivity of $\Rightarrow$ is a theorem of Natural Deduction. It is even easier in the intuitionistic sequent calculus.

$\dfrac{}{\Gamma, A \to A}\,\mathrm{id}$	
$\dfrac{}{\Gamma \to \top}\,\top$	$\dfrac{}{\Gamma, \bot \to A}\,\bot$
$\dfrac{\Gamma, A, B \to C}{\Gamma, A \wedge B \to C}\,\wedge_L$	$\dfrac{\Gamma \to A \quad \Gamma \to B}{\Gamma \to A \wedge B}\,\wedge_R$
$\dfrac{\Gamma, A \to C \quad \Gamma, B \to C}{\Gamma, A \vee B \to C}\,\vee_L$	$\dfrac{\Gamma \to A}{\Gamma \to A \vee B}\,\vee_R$ $\dfrac{\Gamma \to B}{\Gamma \to A \vee B}\,\vee_R$
$\dfrac{\Gamma \to A \quad \Gamma, B \to C}{\Gamma, A \Rightarrow B \to C}\,\Rightarrow_L$	$\dfrac{\Gamma, A \to B}{\Gamma \to A \Rightarrow B}\,\Rightarrow_R$
$\dfrac{\Gamma \to A}{\Gamma, \neg A \to C}\,\neg_L$	$\dfrac{\Gamma, A \to \bot}{\Gamma \to \neg A}\,\neg_R$

Table 3: The rules of inference for LJ, the intuitionistic sequent calculus. The right hand side of each sequent is a single formula.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{B \Rightarrow C, A \rightarrow A}\,\text{id} \qquad \dfrac{\dfrac{}{A, B \rightarrow B}\,\text{id} \quad \dfrac{}{A, B, C \rightarrow C}\,\text{id}}{B \Rightarrow C, A, B \rightarrow C}\,{\Rightarrow_L}}{A \Rightarrow B, B \Rightarrow C, A \rightarrow C}\,{\Rightarrow_L}}{A \Rightarrow B, B \Rightarrow C \rightarrow A \Rightarrow C}\,{\Rightarrow_R}}{A \Rightarrow B \rightarrow (B \Rightarrow C) \Rightarrow (A \Rightarrow C)}\,{\Rightarrow_R}}{\rightarrow (A \Rightarrow B) \Rightarrow ((B \Rightarrow C) \Rightarrow (A \Rightarrow C))}\,{\Rightarrow_R}
$$

The deduction may appear complicated when reading from top to bottom, but there are very few choices. Reading the other way, from bottom to top, we see that there are only one or two connectives—hence one or two rules—that may be used in the next step upward.

The number of possibilities for one step in a deduction of $\Gamma \rightarrow \Delta$ is limited. For each formula in Γ and Δ, there is at most two rules that can be applied. One is the identity rule, and the other is a rule for a connective. A sequent calculus is therefore a good candidate for computer implementation of deduction.

Example 3.10 Another theorem we saw in Natural Deduction was the proof of B from $A \vee B$ and $\neg A$. Here we present a proof of $(A \vee B) \wedge \neg A \Rightarrow B$ as a theorem of the intuitionistic sequent calculus.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \rightarrow A}\,\text{id}}{\neg A, A \rightarrow B}\,{\neg_L} \qquad \dfrac{}{\neg A, B \rightarrow B}\,\text{id}}{A \vee B, \neg A \rightarrow B}\,{\vee_L}}{(A \vee B) \wedge \neg A \rightarrow B}\,{\wedge_L}}{\rightarrow (A \vee B) \wedge \neg A \Rightarrow B}\,{\Rightarrow_R}
$$

Reading from bottom to top, the only choice that has to be made is to choose between $\neg_L$ and $\vee_L$ in the middle line, $A \vee B, \neg A \rightarrow B$.

A feature of a sequent calculus is that every formula that appears in a derivation of Φ is either a subformula of Φ or the negation of a subformula of Φ. In contrast, derivations of Φ in the Hilbert system might contain long formulas whose connection to Φ is not at all clear. The proof of the fact about a sequent calculus is known as the Cut-Elimination Theorem and is an important result in the field of proof theory. It was proved in the 1930's by Gerhard Gentzen, who invented the sequent calculus as a way of studying the nature of deductions.

The **classical sequent calculus** is easy to construct from the intuitionistic version. The rules for the classical calculus, called LK, appear in Table 4. For most of the

$\dfrac{}{\Gamma, A \to \Delta, A}\,\text{id}$	
$\dfrac{}{\Gamma \to \Delta, \top}\,\top$	$\dfrac{}{\Gamma, \bot \to \Delta}\,\bot$
$\dfrac{\Gamma, A, B \to \Delta}{\Gamma, A \wedge B \to \Delta}\,\wedge_L$	$\dfrac{\Gamma \to \Delta, A \quad \Gamma \to \Delta, B}{\Gamma \to \Delta, A \wedge B}\,\wedge_R$
$\dfrac{\Gamma, A \to \Delta \quad \Gamma, B \to \Delta}{\Gamma, A \vee B \to \Delta}\,\vee_L$	$\dfrac{\Gamma \to \Delta, A, B}{\Gamma \to \Delta, A \vee B}\,\vee_R$
$\dfrac{\Gamma \to \Delta, A \quad \Gamma, B \to \Delta}{\Gamma, A \Rightarrow B \to \Delta}\,\Rightarrow_L$	$\dfrac{\Gamma, A \to \Delta, B}{\Gamma \to \Delta, A \Rightarrow B}\,\Rightarrow_R$
$\dfrac{\Gamma \to \Delta, A}{\Gamma, \neg A \to \Delta}\,\neg_L$	$\dfrac{\Gamma, A \to \Delta}{\Gamma \to \Delta, \neg A}\,\neg_R$
$\dfrac{\Gamma \to \Delta, A, B \quad \Gamma, A, B \to \Delta}{\Gamma, A \equiv B \to \Delta}\,\equiv_L$	$\dfrac{\Gamma, A \to \Delta, B \quad \Gamma, B \to \Delta, A}{\Gamma \to \Delta, A \equiv B}\,\equiv_L$

Table 4: The rules of inference for LK, the classical sequent calculus. The right hand side of each sequent can have more than a single formula.

rules of inference, we simply allow more than one formula on the right side of each sequent. A more significant change is the rule for $\vee_R$. To conclude $\Gamma \rightarrow A \vee B$ in the intuitionistic version, we must prove $\Gamma \rightarrow A$ or $\Gamma \rightarrow B$, while in the classical version we have to prove only $\Gamma \rightarrow A, B$. All the intuitionistic rules are special cases of the classical rules, so an intuitionistic proof is classically correct.

Example 3.11 The law of the excluded middle is one result that we expect to be provable in a classical system but not an intuitionistic one.

$$\dfrac{\dfrac{\dfrac{}{A \rightarrow A}\,\text{id}}{\rightarrow A, \neg A}\,\neg_R}{\rightarrow A \vee \neg A}\,\vee_R$$

Where is a non-intuitionistic, classical rule used in the derivation? What goes wrong when you try to construct an intuitionistic derivation?

Example 3.12 One direction of the equivalence of contrapositives is intuitionistically provable. Carefully study the derivation, especially the application of the $\neg_L$ rule.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \rightarrow A}\,\text{id} \quad \dfrac{}{A, B \rightarrow B}\,\text{id}}{A \Rightarrow B, A \rightarrow B}\,\Rightarrow_L}{A \Rightarrow B, \neg B, A \rightarrow \bot}\,\neg_L}{A \Rightarrow B, \neg B \rightarrow \neg A}\,\neg_R}{A \Rightarrow B \rightarrow \neg B \Rightarrow \neg A}\,\Rightarrow_R}{\rightarrow (A \Rightarrow B) \Rightarrow (\neg B \Rightarrow \neg A)}\,\Rightarrow_R$$

The other direction of the contrapositive is classically, but not intuitionistically, provable.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\neg B, A \rightarrow A}\,\text{id}}{\neg B \rightarrow A, \neg A}\,\neg_R \quad \dfrac{\dfrac{}{B \rightarrow \neg A, B}\,\text{id}}{B, \neg B \rightarrow \neg A}\,\neg_L}{\neg B, A \Rightarrow B \rightarrow \neg A}\,\Rightarrow_L}{A \Rightarrow B \rightarrow \neg B \Rightarrow \neg A}\,\Rightarrow_R}{\rightarrow (A \Rightarrow B) \Rightarrow (\neg B \Rightarrow \neg A)}\,\Rightarrow_R$$

Another contrasting pair of formulas are the introduction and elimination of double negation. The introduction of double negation is acceptable in the intuitionistic system. In other words, $A \Rightarrow \neg\neg A$ is an intuitionistic theorem. But the elimination of double negation, $\neg\neg A \Rightarrow A$, has only a classical proof.

3.5 Satisfaction for Sequents

Recall that a valuation is a function $v : \mathsf{Prop} \to \{\text{False}, \text{True}\}$. In the last set of notes, we defined what it means for a valuation v to satisfy a single formula. We extend that definition to sequents by saying that v **satisfies** the sequent $\Gamma \to \Delta$ if v makes either one of the formulas in Γ false or one of the formulas in Δ true.

The sequent $\Gamma \to \Delta$ is **satisfiable** if *some* valuation satisfies it. It is **valid** if *every* valuation satisfies it.

The sequent $\Gamma \to \Delta$ is **falsifiable** if it is not valid, or equivalently, if some valuation makes every formula in Γ true and every formula in Δ false. A sequent is valid if and only if it is not falsifiable.

3.6 Soundness

A typical rule of inference in a sequent calculus has zero or more premises and a conclusion.

$$\frac{\text{Premises}}{\text{Conclusion}}\,\text{Rule}$$

One desirable property for a rule of inference is the following:

> Any valuation that satisfies all of the premises of a rule of inference also satisfies the conclusion.

One can verify directly (do it!) that the property holds for all the rules of inference in the intuitionistic and classical propositional sequent calculi. Some rules, like the id-rule, have no premises, and the conclusions of those rules are always valid.

Working from top to bottom in a correct proof, we see that each sequent in the proof is valid. It follows that the final conclusion of the proof is valid.

Theorem 3.1 (Soundness Theorem) *Every theorem of the intuitionistic or classical sequent calculus is a valid sequent.*

3.7 Completeness

The converse of the property in the previous section can be stated like this:

> Any valuation that satisfies the conclusion of a rule of inference also satisfies all of the premises.

Another way to put it is as a statement about falsifiability:

> Any valuation that falsifies one of the premises of a rule of inference also falsifies the conclusion.

You should be sure that you understand why the two statements are equivalent. The property does not hold for all the rules in the intuitionistic sequent calculus.

Example 3.13 For the intuitionistic rule $\dfrac{\Gamma \to A}{\Gamma \to A \lor B}\lor_R$ it is possible for the premise to be false while the conclusion is true. An extreme example is when A is $\bot$ and B is $\top$.

Example 3.14 The intuitionistic rule $\dfrac{\Gamma \to A}{\Gamma, \neg A \to C}\neg_L$ also does not have the stated property. There may be a valuation that makes A false and C true. Such a valuation will falsify the premise but satisfy the conclusion.

One *can* verify the property (again, do it!) for all the rules of the classical sequent calculus. From this observation, we see that if a conclusion in a classical proof is valid, then all the premises of the proof are also valid. This holds for entire deductions, not just single steps.

The classical, propositional sequent calculus is a very special system. It has three important properties. First, as was just stated, if the conclusion of a proof is valid, then so are the premises.

Second, if a sequent $\Gamma \to \Delta$ contains a formula with a connective, then there is an application of one of the rules of inference which has $\Gamma \to \Delta$ as its conclusion. Moreover, every premise of the rule will have one fewer connective than $\Gamma \to \Delta$. One can verify that by looking at the list of rules. There are rules to eliminate each kind of connective, on the left or the right of the arrow. The premises of such rules have do not contain the corresponding connective, and they do not introduce new ones.

Third, if a sequent contains no connectives, then either it is a consequence of the id, $\bot$, and $\top$ rules, or else it is (obviously) falsifiable. A sequent that contains no connectives has atomic formulas on either side of the arrow. If the same letter appears on the left and the right, then the sequent is provable by the id-rule. If $\bot$ appears on the left or $\top$ on the right, then the sequent is provable by the $\bot$ or $\top$ rules. Otherwise, no letter appears on both left and right, and we can read off a valuation making all the letters on the left true and all those on the right false.

We can use these three properties to try to construct (from bottom to top) a derivation of a given sequent $\Gamma \rightarrow \Delta$. As long as there are premises with connectives, we can continue applying rules. Since each step eliminates one connective, no branch in the proof tree will be longer than the number of connectives in $\Gamma \rightarrow \Delta$, and the process will end after a finite number of steps. At that point, the only premises left will be those without connectives. There are two possibilities:

- All these premises are derivable from the id, $\bot$, and $\top$ rules, and we have constructed a correct derivation of $\Gamma \rightarrow \Delta$.
- One (or more) of the premises is falsifiable, and therefore the conclusion is falsifiable.

We have either a correct derivation of $\Gamma \rightarrow \Delta$ or a valuation that falsifies it. If the sequent is valid, then we must have constructed a derivation. This observation gives us the completeness theorem for the classical sequent calculus.

Theorem 3.2 (Completeness Theorem) *A valid sequent is a theorem of the classical sequent calculus.*

A consequence of the completeness theorem is the fact that every intuitionistically provable sequent is also classically provable. That fact is not obvious because the intuitionistic rules are not, in general, special cases of the classical rules. To demonstrate that fact, suppose that $\Gamma \rightarrow \Delta$ is an intuitionistic theorem. Then by the soundness theorem, $\Gamma \rightarrow \Delta$ is valid. By the completeness theorem, $\Gamma \rightarrow \Delta$ is a theorem of the classical system.

3.8 Completeness for the Hilbert System

There is a mismatch between the theorems of the classical sequent calculus and the theorems of the Hilbert system. The former are sequents, while the latter are single formulas. However, it is still possible to translate a sequent proof to a Hilbert-style proof.

Let us define a sequent $\Phi_0, \Phi_1, \ldots, \Phi_{m-1} \rightarrow \Psi_0, \Psi_1, \ldots, \Psi_{n-1}$ to be **Hilbert-provable** if there is a proof in the Hilbert system of $\Psi_0 \vee \Psi_1 \vee \ldots \vee \Psi_{n-1}$ from the hypotheses Φ_0 through Φ_{m-1}. Implicitly, we are assuming that the formulas are translated into equivalent forms using only the connectives $\neg$ and $\Rightarrow$. Further, the constants $\bot$ and $\top$, which do not appear in Hilbert formulas, must be replaced by $p \wedge \neg p$ and $p \vee \neg p$, respectively.

Lemma 3.3 *I If, in an instance of a rule of inference in the classical sequent calculus, the premises are Hilbert-provable, then the conclusion is Hilbert-provable.*

Using Lemma 3.3 inductively yields a theorem about the Hilbert-provability of sequents.

Theorem 3.4 *If the sequent $\Gamma \rightarrow \Delta$ is provable in the classical sequent calculus, then $\Gamma \rightarrow \Delta$ is Hilbert-provable.*

Corollary 3.5 *If Φ is a valid formula, then Φ is a theorem of the Hilbert system.*

PROOF: If Φ is a valid formula, then $\rightarrow \Phi$ is a valid sequent and is a theorem of the classical sequent calculus, by Theorem 3.2. The sequent $\rightarrow \Phi$ is Hilbert-provable, which for this simple sequent simply means that Φ is a theorem of the Hilbert system.

In section 3.3, we observed that the theorems of the Natural Deduction system are exactly the same as the theorems of the Hilbert System. Consequently, we have established the Completeness Theorem for the Natural Deduction system as well.

3.9 Decidability

The completeness theorem gives us a decision procedure for validity. Starting with a sequent $\Gamma \rightarrow \Delta$, one can apply the classical rules and either end up with a proof of $\Gamma \rightarrow \Delta$ or a valuation that makes $\Gamma \rightarrow \Delta$ false. Viewed in this way, working from the bottom up, building a proof in the classical sequent calculus is a search for a falsifying valuation. The surprising fact is that if we fail to find a falsifying valuation, we have constructed a proof that there is no such falsifying valuation.

Unfortunately, this new decision algorithm for validity may not be much more efficient than looking at truth tables. The reason is that the number of premises can double as we go up one level in the proof.

One consequence of our observations is that it makes no difference in which order we apply the rules. If a sequent is classically provable, any choices we make will result in a proof.

Example 3.15 The order in which we apply rules in the intuitionistic sequent calculus *does* matter. Consider the intuitionistic theorem $A \vee B \rightarrow B \vee A$. If, son-

structing from the bottom up, we choose first to apply $\vee_R$, we are stuck.

$$\dfrac{\dfrac{\dfrac{}{B \to B}\text{id} \qquad A \to B}{A \vee B \to B}\vee_L}{A \vee B \to B \vee A}\vee_R$$

There is nothing that can be done with the remaining premise $A \to B$. A similar problem arises if at the first step we choose the other alternative for $\vee_R$. The theorem *is* provable, however, if we choose $\vee_L$ as the bottom step.

4 Resolution

4.1 Conjunctive Normal Form

A formula Φ is in **conjunctive normal form**, abbreviated CNF, if Φ is of the form

$$C_1 \wedge C_2 \wedge \ldots \wedge C_n$$

where each C_i is of the form

$$\ell_1^i \vee \ell_2^i \vee \ldots \vee \ell_{m_i}^i$$

and ℓ_j^i is either a proposition letter or the negation of a proposition letter. The subformulas C_i are called **clauses** and the subformulas ℓ_j^i are called **literals**.

Every formula is equivalent to a CNF formula. The steps in reducing a formula to CNF are as follows:

1. Eliminate all occurrences of $\Rightarrow$, $\Leftarrow$, and $\equiv$ by replacing them with equivalent constructions using only $\wedge$, $\vee$, and $\neg$. For this, we use the equivalences

   $A \Rightarrow B \leftrightarrow \neg A \vee B$,
   $A \Leftarrow B \leftrightarrow A \vee \neg B$, and
   $A \equiv B \leftrightarrow (A \wedge B) \vee (\neg A \wedge \neg B)$.

2. Push all the negation symbols through the conjunctions and disjunctions, using the DeMorgan laws. Eliminate double negations.

3. Use the distributive laws for $\wedge$ and $\vee$ to push the $\vee$ connectives deeper into the formula.

Along the way, one can eliminated repeated clauses and carry out other simplifications. In general, the resulting CNF formula is much longer than the original formula.

Example 4.1 Consider the negation of an instance of Axiom 2 of the Hilbert system: $\neg\,((p \Rightarrow (q \Rightarrow r)) \Rightarrow ((p \Rightarrow q) \Rightarrow (p \Rightarrow r)))$. We can compute equivalent formulas as follows.

$$\begin{array}{l}(p \Rightarrow (q \Rightarrow r)) \wedge \neg((p \Rightarrow q) \Rightarrow (p \Rightarrow r)) \\ (\neg p \vee (q \Rightarrow r)) \wedge ((p \Rightarrow q) \wedge \neg(p \Rightarrow r)) \\ (\neg p \vee \neg q \vee r) \wedge (\neg p \vee q) \wedge (p) \wedge (\neg r)\end{array}$$

The last line is a CNF formula with four clauses.

The connectives in a CNF formula appear in a regular pattern, and it is not necessary to write them. We think of a clause as a set of literals, implicitly $\vee$-ed together, and a CNF formula as a set of clauses, implicitly $\wedge$-ed together. Thus the CNF formula

$$(\ell_1^1 \vee \ell_2^1 \vee \ldots \vee \ell_{m_1}^1) \wedge (\ell_1^2 \vee \ell_2^2 \vee \ldots \vee \ell_{m_2}^2) \wedge \ldots \wedge (\ell_1^n \vee \ell_2^n \vee \ldots \vee \ell_{m_n}^n)$$

could also be written

$$\left\{\{\ell_1^1, \ell_2^1, \ldots, \ell_{m_1}^1\}, \{\ell_1^2, \ell_2^2, \ldots, \ell_{m_2}^2\}, \ldots, \{\ell_1^n, \ell_2^n, \ldots, \ell_{m_n}^n\}\right\}.$$

We will use the language of formulas and sets interchangeably, speaking of a CNF formula as a set of clauses and a clause as a set of literals.

Recall that the literals ℓ_j^i are either proposition letters or the negations or proposition letters. If ℓ is a literal, then its negation is also a literal and can be obtained by either appending or deleting the negation connective $\neg$. The negation of the literal ℓ is written ℓ^C.

A clause is a disjunction of literals. A clause C is satisfiable if there is a valuation that makes one (or more) of the literals in C true. A clause is falsified by a valuation that makes *all* the clause's literals false. The **empty clause**, denoted $\square$, has no literals and is not satisfiable.

A formula, viewed here as a set of clauses, is satisfiable if there is a valuation that (mutually) satisfies all the clauses in the set.

4.2 Resolution Rule

Two clauses C_a and C_b **clash** on the literal ℓ if ℓ is in C_a and ℓ^C is in C_b. The **resolvent** of clauses C_a and C_b which clash on ℓ is the clause

$$(C_a \setminus \{\ell\}) \cup \left(C_b \setminus \{\ell^C\}\right).$$

The resolution rule is a way of deriving one set of clauses from another. Suppose that S is a set of clauses and that C_a and C_b are clauses in S that clash on ℓ. Let C_r be the resolvent of C_a and C_b. The **resolution rule** tells us that we may add C_r to the set S, obtaining a new set S',

$$S' = S \cup \{C_r\}.$$

Often, there are many possible applications of the resolution rule. There may be several pairs of clashing clauses, and a pair of clashing clauses may clash on several different literals.

The important fact about the resolution rule is that it preserves satisfiability.

Lemma 4.1 *Suppose that S is a set of clauses and that S' arises from S by an application of the resolution rule. A valuation v satisfies all the clauses in S if and only if v satisfies all the clauses in S'.*

PROOF: Notice that S is a subset of S'. If v satisfies all the clauses in S', then v automatically satisfies all the clauses in S.

Conversely, suppose that v satisfies all the clauses in S. Suppose further that the resolution rule was applied to clashing clauses C_a and C_b of S and that the resolvent was C_r. Let ℓ be the literal on which C_a and C_b clash. The only clause in S' that is not in S is the resolvent C_r.

If v makes ℓ true and ℓ^C false, then there must be some literal, different from ℓ^C, in C_b that is made true by v. Call it ℓ'. Then ℓ' is an element of $C_b \setminus \{\ell^C\}$ and hence is an element of C_r. Therefore, C_r is satisfied by v, and v satisfies every clause in S'.

The other possibility, that v makes ℓ false and ℓ^C true, can be handled similarly to complete the proof of the lemma.

4.3 Resolution Procedure

The **resolution procedure** is a way of checking satisfiability. The process is to apply repeatedly the resolution rule. Starting with a set S of clauses, we let $S_0 = S$ and let S_{k+1} be the result of applying the resolution rule to S_k. The process terminates in one of two ways.

- If the resolvent is ever the empty clause $\square$, then declare the set S to be unsatisfiable and terminate.
- If there is no way to choose clashing clauses so that S_{k+1} is different from S_k, then declare the set S to be satisfiable and terminate.

At each step, a new clause is added to the set. The process *will* eventually terminate, because there is a finite number of clauses that can be constructed from the literals in S. But that finite number is exponentially large, and the resolution procedure could take a very long time.

Notice also that the process of searching for clashing clauses can be very inefficient. There is plenty of room for improvement of the running time, but first we have to be concerned with the correctness of the resolution procedure.

Using Lemma 4.1 inductively, we see that S is satisfiable if and only if S_k is satisfiable. If the resolution procedure ever adds the empty clause $\square$ to S_k, then we know that S_k, and hence S, is unsatisfiable. Put another way, if S is satisfiable, then $\square$ will never emerge as a product of the resolution rule. The production of $\square$ is called a **refutation** of the CNF formula S. The observation that a satisfiable formula will never be refuted is a form of soundness for the resolution procedure.

Theorem 4.2 (Soundness for Resolution) *If S is a refutable set of clauses, then S is an unsatisfiable set of clauses.*

Example 4.2 Consider the CNF formula from Example 4.1, written here in set form.

0. Let $S_0 = \Big\{\underline{\{\bar{p}, \bar{q}, r\}}, \underline{\{\bar{p}, q\}}, \{p\}, \{\bar{r}\}\Big\}$.
1. The first and second clauses clash on q, and give $\{\bar{p}, r\}$ as the resolvent. We underline, in the formula above, our choice of clashing clauses. $S_1 = \Big\{\{\bar{p}, \bar{q}, r\}, \{\bar{p}, q\}, \underline{\{p\}}, \{\bar{r}\}, \underline{\{\bar{p}, r\}}\Big\}$.
2. The new resolvent clashes with $\{p\}$ and gives $\{r\}$ as the next resolvent. $S_2 = \Big\{\{\bar{p}, \bar{q}, r\}, \{\bar{p}, q\}, \{p\}, \underline{\{\bar{r}\}}, \{\bar{p}, r\}, \underline{\{r\}}\Big\}$.

3. Finally, $\{r\}$ and $\{\bar{r}\}$ clash and yield the empty resolvent $\square$.

From this refutation, we conclude that the original formula is unsatisfiable. It is not surprising since we began with a formula that is equivalent to the negation of a valid formula.

In this application of resolution, we used all the clauses of the original formula. If any one were omitted, the formula would be satisfiable and hence *not* refutable. For instance, the set $\{\{\bar{p}, \bar{q}, r\}, \{\bar{p}, q\}, \{p\}\}$ contains the first three clauses from S_0 and is satisfied by making p, q, and r all true. What is the result of applying the resolution procedure to that set?

At each step in the example, we had several pairs of clashing clauses. Different choices would have produced a different, and perhaps longer, sequence of steps.

4.4 Completeness for Resolution

The flip side to soundness is completeness. We can ask, "Will every non-satisfiable formula be refuted?" Or even more strongly, we might wonder if we can find a satisfying valuation: "If the resolution procedure is started on S and terminates without producing $\square$, do we have enough information to construct a valuation that satisfies S?" The answer to both questions is "yes."

We begin our discussion with the tree of possible valuations on some finite set of proposition letters $\{p_1, p_2, \ldots, p_s\}$. It is a full binary tree of height s in which each node represents a partial assignment of false or true to some of the letters. We write partial assignments as strings over the characters p_1 through p_s and $\bar{p}_1$ through $\bar{p}_s$. At the root is the empty assignment which gives truth values to none of the letters. A child makes a truth assignment to one more letter than does its parent. The two children of a parent represent the assignment of false and true to the new letter. There are 2^s leaves, one for each possible valuation. The tree for three letters is illustrated in Table 5.

A node in the valuation tree is a partial assignment of truth values. Such a partial assignment may be sufficient to determine the truth values of certain formulas. We can speak of one partial assignment "extending" another. The complete assignments are at the leaves.

Let S be a set of clauses, and let S_k be the last set constructed by the resolution procedure. A clause C **covers** a node n in the valuation tree if it satisfies the following properties.

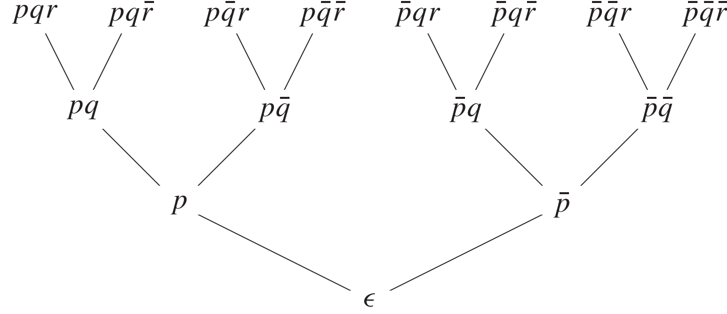

Table 5: The tree of possible valuations on the three proposition letters p, q, and r.

1. The clause C is an element of S_k.
2. The proposition letters that appear in C are among the letters which are given truth values by n.
3. The clause C is falsified by every assignment that extends n.

Some nodes may be covered by many clauses, while other nodes may be uncovered. By property 2, the empty clause $\Box$ is the only clause that can cover the root.

Notice that we are not describing an algorithm here, we are simply considering a valuation tree with all the possible covering clauses.

Lemma 4.3 *If a non-leaf node is covered, then both its children are covered.*

PROOF: Suppose that clause C covers a non-leaf node. One can easily check that C satisfies all the conditions to cover the node's children.

Lemma 4.4 *If both children of a node are covered, then the node itself is covered.*

PROOF: Suppose that B is a node with children Bq and $B\bar{q}$, and C_0 and C_1 are clauses covering the nodes Bq and $B\bar{q}$, respectively. If the proposition letter q does not appear in C_0, then C_0 covers B. Similarly, if q does not appear in C_1, then C_1 covers B. If q appears in both clauses, then $\neg q$ must be in C_0, and q must be in C_1, because the partial valuation makes *all* the elements of covering clauses false. Therefore, the clauses C_0 and C_1 clash on q, and the resolvent C_r is

$$(C_0 \setminus \{\bar{q}\}) \cup (C_1 \setminus \{q\})\,. \tag{1}$$

We claim that C_r covers B. There are three properties to verify. For the first property, the resolvent C_r is in the set S_k because the resolution procedure will not terminate until all resolvents are present.

For the second property, notice that a proposition letter appearing in C_0 and C_1 is either q or a proposition letter appearing in B. It is enough to show that the letter q does not appear in the resolvent C_r. We observe that q cannot appear in $C_0 \setminus \{\bar{q}\}$. Otherwise, C_0 would contain both q and $\bar{q}$, and C_0 would be satisfiable by *any* valuation. Such a clause can never cover any node. Similarly, C_1 cannot contain $\bar{q}$. Therefore, the resolvent (1) does not contain the letter q.

For the third property, consider any literal ℓ in $C_0 \setminus \{\bar{q}\}$. That literal is made false by any valuation that extends Bq. But since the literal ℓ is not q or its negation, its truth assignment depends only on the partial valuation B. Therefore every literal in $C_0 \setminus \{\bar{q}\}$ is made false by the truth assignment B. Similarly, every literal in $C_1 \setminus \{q\}$ is made false by the truth assignment B.

Because it satisfies all three properties, the resolvent C_r covers the node B, and the lemma is proved.

Theorem 4.5 (Completeness for Resolution) *If the set S of clauses is unsatisfiable, then S is a refutable set of clauses.*

PROOF: Suppose that S is an unsatisfiable set of clauses, S_k is the last step in the resolution procedure starting from S, and we have a valuation tree with covering clauses from S_k. Since S is unsatisfiable, all the leaves of the tree are covered by elements of S. Using Lemma 4.4 inductively, we conclude that all the nodes of the tree are covered by elements of S_k. In particular, the root must be covered, and $\square$ is the only possible covering clause for the root. Therefore, $\square$ appeared in the resolution procedure, and S is refutable. The completeness theorem has been proved.

There is a way in which we can construct a satisfying valuation when the resolution procedure tells us that there is one. There must be one uncovered leaf in the valuation tree, and all the nodes on the path from the root to that leaf must also be uncovered. There may be many satisfying valuations, and we can find one by following uncovered vertices from the root.

The only difficulty with the preceding sketch is that we have not *constructed* the tree; we simply used it in our proofs. But we can imagine navigating through the tree and do as well. As was just remarked, the root is uncovered. We start with the empty valuation. If p_1 is the first proposition letter to be considered, then we

have to discover whether the node in direction p_1 or the node in direction $\bar{p}_1$ is uncovered. The only clauses available to cover nodes at that level of the tree are the unit clauses. If S_k contains the clause $\{p_1\}$, then the node $\bar{p}_1$ is covered by $\{p_1\}$. The $\bar{p}_1$ direction is excluded, and we make p_1 true. Otherwise, we make p_1 false.

In general, if we have constructed an assignment A_{t-1} to the letters p_1 through p_{t-1}, then (in our imagination) we are at the uncovered node A_{t-1} and we try to extend our assignment to $A_{t-1}p_t$. Consider p_t together with the *complements* of the literals in A_{t-1}. If some subset of these literals appears in S_k, then there is a clause covering $A_{t-1}\bar{p}_t$. In that case, the $\bar{p}_t$ direction is excluded, and we make p_t true. Otherwise, we make p_t false.

Using the inductive step in the last paragraph, we may find a satisfying assignment for S. Our process falls a little short as an algorithm, because it requires that we look for *subsets* of clauses in S_k—a time-consuming prospect.

4.5 Optimizations

The following lemmas can be used to reduce the number of clauses, or the size of the individual clauses, in a set of clauses. These reductions are important, because the resolution procedure is exponential in the number of clauses *and* the number of literals.

If S and T are sets of clauses, we write $S \approx T$ to mean that S is satisfiable if and only if T is satisfiable.

Lemma 4.6 *Suppose that ℓ occurs in some of the clauses of S but ℓ^C does not. Let T be the set of clauses in S which do* not *contain ℓ. Then $S \approx T$.*

Lemma 4.7 *Suppose that S contains a unit clause $\{\ell\}$. Let T be the set of clauses in S which do not contain ℓ. Then $S \approx T$.*

Notice that after applying the result of Lemma 4.7, one is in a position to apply the result of Lemma 4.6 with ℓ^C.

Lemma 4.8 *Suppose that S contains a clause C which contains both a literal and its complement. Then $S \approx S \setminus \{C\}$.*

Lemma 4.9 *Suppose that S contains clauses C_0 and C_1 for which $C_0 \subset C_1$. Then $S \approx S \setminus \{C_1\}$.*

4.6 Resolution and Logical Consequence

If we want to see if a formula Ψ is valid, we just use resolution to check that $\neg\Psi$ is unsatisfiable.

Suppose that we want to see if $\Phi_1, \Phi_2, \ldots, \Phi_n \models \Psi$. This is equivalent to showing that $\Phi_1 \wedge \Phi_2 \wedge \ldots \wedge \Phi_n \Rightarrow \Psi$ is valid. That fact, in turn, is equivalent to the unsatisfiability of $\neg(\Phi_1 \wedge \Phi_2 \wedge \ldots \wedge \Phi_n \Rightarrow \Psi)$, which can be determined by resolution.

One obstacle to applying resolution is that we sometimes encounter formulas that are not CNF formulas. Assuming that the Φ_i's are CNF, the formula at the end of the previous paragraph is close to being CNF.

$$\neg(\Phi_1 \wedge \Phi_2 \wedge \ldots \wedge \Phi_n \Rightarrow \Psi) \leftrightarrow \Phi_1 \wedge \Phi_2 \wedge \ldots \wedge \Phi_n \wedge \neg\Psi$$

It remains only to put $\neg\Psi$ into CNF. Unfortunately, it may take a fair amount of work to put the negation of a CNF formula into CNF.

Example 4.3 We have had several occasions to conclude q from $p \vee q$ and $\neg p$. In the context of resolution, we try to refute the clauses $p \vee q$, $\neg p$, and $\neg q$.

0. Let S_0 be $\{\{p, q\}, \{\neg p\}, \{\neg q\}\}$.
1. The first two clauses clash on p. Their resolvent is $\{q\}$, so we set S_1 to be the set $\{\{p, q\}, \{\neg p\}, \{\neg q\}, \{q\}\}$.
2. The last two clauses in S_1 clash on q. Their resolvent is $\Box$. The resolution procedure terminates with a refutation of S_0.

We can therefore conclude that $p \vee q, \neg p \vdash q$. The same resolution also yields $p \vee q, \neg q \vdash p$ and $\neg p, \neg q \vdash \neg(p \vee q)$.

5 Predicate Logic

The propositional logic, which we have studied until now, is rather weak. It cannot, for example, formalize the assertion

> All one-eyed aliens have one eye.

Nor can it express the deduction

Sir Edmund Hillary climbed Mount Everest.
Sir Edmund Hillary is a human being.
Therefore, some human climbed Mount Everest.

Propositional logic falls short because it has no way of distinguishing *individuals*. The examples above use the words "all" and "some" whose meaning depends on having a collection of objects, not just propositions, under discussion.

Predicate logic fills the deficiency by introducing variables and constants into the underlying language. We will follow a pattern similar our introduction to propositional logic. In the new case, syntax and semantics will be only a little more complicated. When we come to proof systems, we will have to be careful to avoid faulty reasoning.

5.1 Syntax

Consider an alphabet Σ which is the union of eight sets. The first three sets contain the **non-logical symbols.**

1. $\mathcal{A} = \{a_1, a_2, \ldots\}$ is the finite or countably infinite set of **constant symbols.**
2. $\mathcal{F} = \{f_1, f_2, \ldots\}$ is the finite or countably infinite set of **function symbols.**
3. $\mathcal{P} = \{p_1, p_2, \ldots\} \cup \{\doteq\}$ is the finite or countably infinite set of **predicate symbols.**

There are two functions $\text{arity}_{\mathcal{F}} : \mathcal{F} \to \mathbb{N}^+$ and $\text{arity}_{\mathcal{P}} : \mathcal{P} \to \mathbb{N}$ which determine the number of parameters taken by each function and predicate symbol. The special symbol $\doteq$ is called the **equality symbol** and always has an arity of 2.

Note that predicate symbols can have arity zero, but function symbols cannot. The reason is that 0-ary functions are just constants, and it is more natural to consider constants separately from functions.

The three sets $\mathcal{A}$, $\mathcal{F}$, and $\mathcal{P}$, together with the arity-functions, form the **non-logical** part of our language. These components will vary from case to case, depending on what the language is describing. We will use the symbol $\mathcal{L}$ for the non-logical part of our language.

The part of the language that is the same in all cases is the **logical** part. There are five more sets containing **logical symbols.**

4. $\mathcal{X} = \{x_1, x_2, \ldots\}$ is the (countably infinite) set of **variables.**

5. $\mathcal{C} = \{\neg, \wedge, \vee, \Rightarrow, \Leftarrow, \equiv\}$ is the set of **connectives.**

6. $\mathcal{V} = \{\top, \bot\}$ is the set of **truth constants.**

7. $\mathcal{Q} = \{\forall, \exists\}$ is the set of **quantifiers.**

8. $\mathcal{S} = \{), ,, (\}$ is the set of **auxiliary symbols.**

Instead of using subscripts we often write

$a, b, \ldots$ for constant symbols,

$f, g, \ldots$ for function symbols,

$p, q, \ldots$ for predicate symbols, and

$x, y, \ldots$ for variables.

As is by now customary, we define a set of **terms** by structural induction.

1. Every constant symbol is a term.
2. Every variable is a term.
3. If f is a function symbol with $\text{arity}_{\mathcal{F}}(f) = n$ and $t_1, t_2, \ldots, t_n$ are terms, then $f(t_1, t_2, \ldots, t_n)$ is a term.

Examples of terms are a, $f(x, c)$, and $g(h(a, f(b)), g(y, z))$.

The set of **atomic formulas** (or atoms) is also defined by structural induction.

1. The constants $\bot$ and $\top$ are atomic formulas.
2. If p is a predicate symbol with $\text{arity}_{\mathcal{P}}(p) = n$ and $t_1, t_2, \ldots, t_n$ are terms, then $p(t_1, t_2, \ldots, t_n)$ is an atomic formula.

Observe that functions can be nested in terms, but predicates cannot be nested in atomic formulas. Examples of atomic formulas are $p(a)$, $q(f(x, c), y)$, and $r(a, g(h(a, f(b)), g(y, z)), f(z))$.

The set of **well-formed formulas** (or just formulas or wffs) is again defined by structural induction.

1. Every atomic formula is a wff.
2. If Φ is a wff, then $(\neg\Phi)$ is a wff.

3. If Φ and Ψ are wffs and $\diamond$ is one of the binary connectives $\wedge$, $\vee$, $\Rightarrow$, $\Leftarrow$, or $\equiv$, then $(\Phi \diamond \Psi)$ is a wff.

4. If Φ is a wff and x is a variable, then $(\forall x\ \Phi)$ and $(\exists x\ \Phi)$ are wffs.

The rules for eliminating parentheses around connectives apply in predicate logic as well as the propositional logic. We can also eliminate the parentheses associated with quantifiers when a formula starts with a string of two or more quantifiers. An example of a wff is

$$\forall y\, \exists z\, (r(a, g(h(a, f(b)), g(y, z)), f(z)) \wedge \forall x\, (p(x))).$$

5.2 Semantics

The uninterpreted formula above is not very interesting. We interpret wffs by giving meaning to the components of the language of predicate logic.

The distinction between syntax and semantics is clearer here than in the propositional logic, because the "meaning" of a wff carries some informational content. It is not just "false" or "true" as it was in the propositional case.

Example 5.1 In everyday speech, we use predicates. The sentence

The son of Sam is a serial killer.

can be translated as

serialKiller(son(Sam)).

where serialKiller is a predicate symbol, son is a function symbol, and Sam is a constant symbol. We interpret these, respectively, as a predicate, a function, and a constant.

It is a bit odd for "son" to be a function, because Sam (or anyone else) may have more than one son. We can replace "son" with a binary predicate "sonOf" and use an existential quantifier.

$\exists k$ (sonOf(k, Sam) $\wedge$ serialKiller(k)).

Example 5.2 The predicate logic was developed to describe mathematical systems. For example, the axioms of linear order can be cast as wffs. Suppose that p is a predicate symbol and $\text{arity}_{\mathcal{P}}(p) = 2$.

1. $\forall x\ p(x, x)$.

2. $\forall x\, \forall y\, \forall z\, (p(x, y) \wedge p(y, z) \Rightarrow p(x, z)$.
3. $\forall x\, \forall y\, (p(x, y) \wedge p(y, x) \Rightarrow x \doteq y)$.
4. $\forall x\, \forall y\, (p(x, y) \vee p(y, x))$.

If we want to assert that there is a largest element, we can do it with the wff

$$\exists z\, (\forall x\, (p(x, z))).$$

Or if we want to assert that the ordering is dense (having an element strictly between any two distinct elements), we can do it with another formula.

$$\forall x\, \forall y\, (p(x, y) \wedge \neg x \doteq y \Rightarrow \exists z\, (p(x, z) \wedge \neg x \doteq z \wedge p(z, y) \wedge \neg z \doteq y)).$$

Notice that we use $\doteq$ as an infix operator, writing $x \doteq y$ instead of $\doteq(x, y)$. One could also replace p with the more natural infix operator $\leq$.

In order to assign meanings to wffs, we must specify

- a set M of individuals (over which the quantifiers and variables will range),
- an element in M for each constant symbol in the language,
- an n-ary function on M for each n-ary function symbol in the language, and
- an n-ary relation on M for each n-ary predicate symbol in the language.

Proceeding formally, we define an $\mathcal{L}$-**structure** to be a pair (M, I) in which M is a non-empty set called the **domain** and I is a function on the non-logical symbols, satisfying the following conditions.

1. If a is a constant symbol, then $I(a)$ is an element of M.
2. If f is a function symbol and $\text{arity}_{\mathcal{F}}(f) = n$, then $I(f) : M^n \to M$.
3. If p is a function symbol other than the equality symbol and $\text{arity}_{\mathcal{P}}(p) = n$, then $I(p) : M^n \to \{\text{False}, \text{True}\}$.

   The equality symbol is always interpreted as the identity predicate: $I(\doteq) : M^2 \to \{\text{False}, \text{True}\}$, where

$$I(\doteq)(x, y) = \begin{cases} \text{True} & \text{if } x = y \text{, and} \\ \text{False} & \text{if } x \neq y. \end{cases}$$

The function I maps constant symbol to constants, function symbols to functions, and predicate symbols to predicates.

We want to assign elements of M to the variables as well, but we cannot do it in the $\mathcal{L}$-structure because we must allow the variables to vary. An **assignment** is a function from the set $\mathcal{X}$ of variables into M. If $s : \mathcal{X} \to M$ is an assignment $s[x_i := a]$ denotes the new assignment given by

$$s[x_i := a](u) = \begin{cases} a & \text{if } u = x_i \text{, and} \\ s(u) & \text{otherwise.} \end{cases}$$

The assignment s gives a value to each variable. The "meaning" of an arbitrary term in an $\mathcal{L}$-structure depends on the assignment, so we think of the meaning as a function from assignments into elements of the domain. Naturally, we use structural induction, mirroring the definition of terms.

1. If a is a constant symbol, then $a_M(s) = I(a)$.
2. If x is a variable, then $x_M(s) = s(x)$.
3. If t is the term $f(t_1, t_2, \ldots, t_n)$, then $t_M(s) = I(f)\,((t_1)_M(s), (t_2)_M(s), \ldots, (t_n)_M(s))$.

Similarly, the "meaning" of a wff is a function from assignments into {False, True}. We begin with atomic formulas.

1. $\bot_M(s) = \text{False}$.
2. $\top_M(s) = \text{True}$.
3. If A is the atomic formula $p(t_1, t_2, \ldots, t_n)$, then $A_M(s) = I(p)\,((t_1)_M(s), (t_2)_M(s), \ldots, (t_n)_M(s))$.

Once we have "meanings" for atomic formula, it is easy to extend it to formulas with the connectives. We just use the truth-table definitions of the connectives.

4. If $\Phi_M(s) = \text{False}$, then $(\neg\Phi)_M(s) = \text{True}$.
   Otherwise, $(\neg\Phi)_M(s) = \text{False}$.
5. If $\Phi_M(s) = \Psi_M(s) = \text{True}$, then $(\Phi \wedge \Psi)_M(s) = \text{True}$.
   Otherwise, $(\Phi \wedge \Psi)_M(s) = \text{False}$.
6. If $\Phi_M(s) = \Psi_M(s) = \text{False}$, then $(\Phi \vee \Psi)_M(s) = \text{False}$.
   Otherwise, $(\Phi \vee \Psi)_M(s) = \text{True}$.

7. If $\Phi_M(s) = \text{True}$ and $\Psi_M(s) = \text{False}$, then $(\Phi \Rightarrow \Psi)_M(s) = \text{False}$. Otherwise, $(\Phi \Rightarrow \Psi)_M(s) = \text{True}$.

8. $(\Phi \Leftarrow \Psi)_M(s) = (\Psi \Rightarrow \Phi)_M(s)$.

9. If $\Phi_M(s) = \Psi_M(s)$, then $(\Phi \equiv \Psi)_M(s) = \text{True}$. Otherwise, $(\Phi \equiv \Psi)_M(s) = \text{False}$.

The really new part here involves the quantifiers. The assignments were introduced for the benefit of this definition.

10. If for *all* m in M, $\Phi_M(s[x := m]) = \text{True}$, then $(\forall x\ (\Phi))_M\ (s) = \text{True}$. Otherwise, $(\forall x\ (\Phi))_M\ (s) = \text{False}$.

11. If for *some* m in M, $\Phi_M(s[x := m]) = \text{True}$, then $(\forall x\ (\Phi))_M\ (s) = \text{True}$. Otherwise, $(\forall x\ (\Phi))_M\ (s) = \text{False}$.

Example 5.3 Let M be any linearly ordered set, as defined earlier in the course. Interpret the 2-ary predicate symbol p to mean $\leq$. If Φ is one of the four axioms given in Example 5.2, then $\Phi_M(s) = \text{True}$ for all assignments s.

Looking explicitly at the reflexive axiom, $\forall x\ p(x, x)$, we see that the atomic formula $p(x, x)$ is true in M with assignment s if

$$(p(x, x))_M\ (s) = \text{True},$$

or equivalently, if

$$I(p)(s(x), s(x)) = \text{True}.$$

Further, $(\forall x\ p(x, x))_M\ (s) = \text{True}$ if for each m from M, $(p(x, x))_M\ (s[x := m]) = \text{True}$. As we just saw, that translates to $I(p)(s[x := m](x), s[x := m](x)) = \text{True}$, or simply $I(p)(m, m) = \text{True}$. Exactly as we expect,

$$(\forall x\ p(x, x))_M\ (s) = \text{True}$$

if and only if

$$\text{for each } m \text{ from } M, I(p)(m, m) = \text{True}.$$

The binary predicate $I(p)$ is a function from M^2 to {False, True}. The same information can be conveyed by specifying those pairs of elements of M which are made True by $I(p)$. We can define a binary relation R_p on the elements of M;

$$R_p = \{(p, q) \in M^2 \mid I(p)(p, q) = \text{True}\}.$$

Then we have the connection that the relation R_p is a reflexive relation on M if and only if the formula $\forall x\ p(x, x)$ is true in the $\mathcal{L}$-structure M for all assignments s. It is common to identify an n-ary predicate with an n-ary relation.

Predicate logic is often called "first-order logic" because it permits quantification over individuals. There are second-order and higher-order logics which permit quantification over sets of individuals and functions on individuals. We will not study those logics in this course.

5.3 Free and Bound Variables

Intuitively, we believe that $\forall x\ p(x)$ and $\forall z\ p(z)$ mean the same thing, even though $p(x)$ without the quantifier may be different from $p(y)$. On the other hand, the formula $\forall x\ (p(x) \wedge \exists z\, r(x, z))$ is *not* equivalent to $\forall z\ (p(z) \wedge \exists z\, r(z, z))$.

Although the formula $\exists x\ (p(x) \wedge \forall x\, q(x))$ displays a poor choice of variables, it is well-formed and is equivalent (in a sense to be defined precisely later) to $\exists z\ (p(z) \wedge \forall y\, q(y))$.

These examples tell us that we have to take care with our choice of variables. One distinction is the difference between free and bound variables. Roughly speaking, x is free in $\exists z\ (\Psi(x, z))$ while z is bound (by the quantifier). Further, x lies in the scope of the binding of z.

Giving formal definitions of these concepts is a bit delicate. It is not enough to speak of variables in a formula; we have to identify specific occurrences of variables in the formula. Any occurrence of a variable is either free or bound, but not both. If it is bound, it is bound to a particular quantifier.

1. Any occurrence of a variable in an atomic formula is free in that formula.

2. Any occurrence of a variable that is free in Φ is also free in $\neg\Phi$. Any occurrence that is bound in Φ is bound (by the same quantifier) in $\neg\Phi$.

3. Let $\diamond$ be one of the binary connectives. Any occurrence of a variable that is free in Φ or in Ψ is also free in $\Phi \diamond \Psi$. Any occurrence of a variable that is bound in Φ or in Ψ is bound (by the same quantifier) in $\Phi \diamond \Psi$.

4. Any free occurrence of a variable, other than x, in Φ is also free in $\forall x\ \Phi$. Any free occurrence of x in Φ is bound to the leading quantifier in $\forall x\ \Phi$. Any bound occurrence of a variable in Φ remains bound by the same quantifier in $\forall\Phi$. The same definitions apply to the existential formula $\exists x\ \Phi$.

A similar definition by structural induction tells us when an occurrence of a variable falls within the scope of a quantifier. An occurrence of a variable may fall within the scope of several quantifiers.

1. Any occurrence of a variable in an atomic formula is not within the scope of any quantifier.

2. If an occurrence of a variable is within the scope of a quantifier in Φ, then it is with the scope of the same quantifier in $\neg\Phi$.

3. Let $\diamond$ be one of the binary connectives. If an occurrence of a variable is within the scope of a quantifier in Φ, then it is in the scope of the same quantifier in $\Phi \diamond \Psi$. Similarly, an occurrence of a variable within the scope of a quantifier in Ψ is in the scope of the same quantifier in $\Phi \diamond \Psi$.

4. Any occurrence of a variable in Φ is within the scope of the outer quantifier in $\forall x\ \Phi$. Any occurrence of a variable that is within the scope of a quantifier in Φ remains in the scope of that quantifier in $\forall x\ \Phi$. The same definitions apply to the existential formula $\exists x\ \Phi$.

Notice that we have taken some liberties by identifying an occurrence of a variable in Φ with the corresponding occurrence in $\Phi \diamond \Psi$. The variable that is, say, the 47th symbol in Φ is considered the same as the variable that is the 47th symbol of $\Phi \diamond \Psi$. The variable that is the 29th symbol in Ψ is considered the same as the variable that is in position $\text{length}(\Phi) + 1 + 29$ in $\Phi \diamond \Psi$. (If we include all the parentheses, the positions change a bit more than indicated here.) To be completely formal in these matters would make the definitions much more complicated.

One consequence of the definitions for interpretations is that the truth of a formula Φ does not depend on *all* of an assignment s; it only depends on the values of s at the free variables of Φ.

Lemma 5.1 *If s and s' are valuations that agree on all the variables in the term t, then $t_M(s) = t_M(s')$.*

PROOF: The proof is by structural induction on terms. If a is a constant symbol, then $a_M(s) = I(a)$. The value of $a_M(s)$ is independent of s, so $a_M(s) = a_M(s')$.

If x is a variable, and s and s' agree on x, then $x_M(s) = s(x) = s'(x) = x_M(s')$.

If $f(t_1, \ldots, t_n)$ is a term composed from a function symbol f, and s and s' agree on all the variables in $f(t_1, \ldots, t_n)$, then s and s' agree on the variables in each t_i.

By the hypothesis of induction, $(t_i)_M(s) = (t_i)_M(s')$ for each i between 1 and n. We then have

$$\begin{aligned}(f(t_1,\ldots,t_n))_M(s) &= I(f)((t_1)_M(s),\ldots,(t_n)_M(s))\\ &= I(f)((t_1)_M(s'),\ldots,(t_n)_M(s'))\\ &= (f(t_1,\ldots,t_n))_M(s'),\end{aligned}$$

as desired.

Lemma 5.2 *If s and s' are valuations that agree on all the free variables of Φ, then $\Phi_M(s) = \Phi_M(s')$.*

The proof of Lemma 5.2 is by induction on formulas. An important special case occurs when there are no free variables. A formula that has no free variables is **closed**.

Corollary 5.3 *If Φ is a closed formula and if s and s' are* any *valuations, then $\Phi_M(s) = \Phi_M(s')$.*

5.4 Satisfiability and Validity

Let M be an $\mathcal{L}$-structure and s an assignment. If $\Phi_M(s) =$ True, then we say that M **satisfies** Φ **with** s. In symbols, we write $M \models \Phi(s)$.

The formula Φ is **satisfiable in** M if, for some assignment s, $M \models \Phi(s)$. The formula Φ is simply **satisfiable** if there is a structure M and an assignment s for which $M \models \Phi(s)$.

The formula Φ is **valid** in M, written $M \models \Phi$, if $M \models \Phi(s)$ for all assignments s. In this case, we also say that M is a **model** of Φ.

The formula Φ is **valid** if it is valid in every $\mathcal{L}$-structure.

The notions of satisfiability and validity are extended to sets of formulas in a natural way. For example, the set of formulas Γ is **mutually satisfiable** if there is an $\mathcal{L}$-structure M and an assignment s such that $M \models \Phi(s)$ for each formula Φ in Γ.

A formula Ψ is a **logical consequence** of a set Γ of formulas if for each $\mathcal{L}$-structure M and assignment s, either $\Psi_M(s) =$ True or else $\Phi_M(s) =$ False for some Φ in Γ. This is a fancy way of saying that Ψ is true whenever all the elements of Γ are true. If Ψ is a logical consequence of Γ, we write $\Gamma \models \Psi$.

Two formulas are **logically equivalent** if either is a logical consequence of the other. It is easy to prove (do it!) that Φ and Psi are logically equivalent if and only if $\Phi \equiv \Psi$ is valid.

5.5 Substitution

The notion of substitution of one term for another seems easy, but it is easy to make errors. A formal definition is actually helpful here.

The notation $\alpha[x := t]$ will stand for the result of substituting the term t for the variable x in the string α. The string α can be a term or a formula. The first case, for a term, is straightforward.

1. $a[x := t] = a$, for a constant symbol a.
2. $x[x := t] = t$, for a variable x.
3. $y[x := t] = y$, for a variable y with $x \neq y$.
4. $f(t_1, \ldots, t_n)[x := t] = f(t_1[x := t], \ldots, t_n[x := t])$, for a function symbol f and terms t_1 through t_n.

Notice that substitution is a purely syntactic operation.

Most of the cases in the definition for substitution in a formula are equally straightforward.

1. $\bot[x := t] = \bot$.
2. $\top[x := t] = \top$.
3. $p(t_1, \ldots, t_n)[x := t] = p(t_1[x := t], \ldots, t_n[x := t])$, for a predicate symbol p and terms t_1 through t_n.
4. $(\neg\Phi)[x := t] = \neg(\Phi[x := t])$.
5. $(\Phi \diamond \Psi)[x := t] = (\Phi[x := t] \diamond \Psi[x := t])$, if $\diamond$ is one of the binary connectives.
6. $(\forall x\ \Phi)[x := t] = (\forall x\ \Phi)$.
7. $(\exists x\ \Phi)[x := t] = (\exists x\ \Phi)$.

Notice the last two cases. Since there are no free occurrences of x in $\forall x\,\Phi$, the substitution does nothing.

One might be tempted to complete the definition like this.

8. $(\forall y\,\Phi)[x := t] = (\forall y\,\Phi[x := t])$, for $x \neq y$. [Incorrect!]

The difficulty is that t may contain the variable y, which is then "captured" by the quantifier. For example, consider the substitution $(\forall y\, p(x, y))[x := y]$. The result, according to the attempted definition is $\forall y\, p(y, y)$, which is not what we expect.

The solution is to change the bound variables y to avoid the conflict. Here are the correct definitions.

8. If $x \neq y$, let z be a variable that does not appear at all in $\forall y\,\Phi$ and in t. Then $(\forall y\,\Phi)[x := t] = (\forall z\,\Phi[y := z][x := t])$.
9. If $x \neq y$, let z be a variable that does not appear at all in $\forall y\,\Phi$ and in t. Then $(\exists y\,\Phi)[x := t] = (\exists z\,\Phi[y := z][x := t])$.

We can make these clauses completely unambiguous by choosing z to be the *first* variable (that is, the variable x_j with the least subscript) that does not occur in $\forall y\,\Phi$ or t.

According to the our definition, the result of the substitution $(\forall y\, p(x, y))[x := y]$ is $\forall z\, p(y, z)$.

5.6 First-order Natural Deduction

So far, we have the semantic notions of satisfaction, truth, and validity, but we have not introduced any syntactic proof systems. One such system is the Natural Deduction system. Start with all the rules for the propositional system, and add the following four to cover the quantifiers.

1. $\dfrac{\Phi}{\forall x\,\Phi}\forall_I$, provided that x does not occur as a free variable in any of the undischarged assumptions on which Φ depends.
2. $\dfrac{\forall x\,\Phi}{\Phi[x := t]}\forall_E$.
3. $\dfrac{\Phi[x := t]}{\exists x\,\Phi}\exists_I$.

4. $$\dfrac{\exists x\ \Phi \quad \begin{array}{c}\not{\Phi}\\ \vdots \\ \Psi\end{array}}{\Psi}\exists_E$$, provided that x does not occur as a free variable in Ψ or in any of the remaining undischarged assumptions on which Ψ depends. Here, we consider Φ to be discharged so that x may occur as a free variable in Φ.

These rules of the Natural Deduction system really are "natural." For example, the informal version of the $\forall_I$ rule is

> If we can prove Φ without making any assumptions about x, then we can conclude $\forall x\ \Phi$.

The conditions on the $\forall_I$ and $\exists_E$ rules are important; see Example 5.10.

Example 5.4 One special case of $\forall_E$ is when t is the same as x. Then we have just $\dfrac{\forall x\ \Phi}{\Phi}\forall_E$. If x does not occur as a free variable in Φ we can reason in the opposite direction; $\dfrac{\Phi}{\forall x\ \Phi}\forall_I$. From these two inferences, we conclude $(\forall x\ \Phi) \equiv \Phi$ whenever x does not occur free in Φ. This result is consistent our intuition that quantifying over a variable that does not appear in a formula does not change the meaning of a formula.

Example 5.5 The rule $\exists_I$ also has a degenerate case when t is x. We see it in the inference which proves the distribution of $\exists$ over $\wedge$.

$$\dfrac{\exists x\ (\Phi \wedge \Psi) \qquad \dfrac{\dfrac{\dfrac{\Phi \wedge \Psi}{\Phi \quad\quad \Psi}\wedge_E}{\dfrac{\Phi}{\exists x\ \Phi}\exists_I \quad \dfrac{\Psi}{\exists x\ \Psi}\exists_I}}{\exists x\ \Phi \wedge \exists x\ \Psi}\wedge_I}{\exists x\ \Phi \wedge \exists x\ \Psi}\exists_E$$

Invoke $\Rightarrow_I$ and discharge the hypothesis to obtain the theorem

$$\exists x\ (\Phi \wedge \Psi) \Rightarrow (\exists x\ \Phi \wedge \exists x\ \Psi).$$

We would not expect to be able to prove the converse. Why?

Example 5.6 Let C and H be a unary predicate symbols. We will think of $C(x)$ as meaning x climbed Mount Everest, and $H(x)$ as meaning x is a human being.

Let e be a constant symbol denoting Sir Edmund Hillary. Then the deduction at the beginning of this set of notes can be rendered in Natural Deduction.

$$\dfrac{\dfrac{C(e) \quad H(e)}{C(e) \wedge H(e)}\wedge_I}{\exists x\ (C(x) \wedge H(x))}\exists_I$$

We note, in the application of the $\exists_I$ rule, that $C(e) \wedge H(e)$ is the same formula as $(C(x) \wedge H(x))\,[x := e]$.

Example 5.7 One important result is the distribution of $\forall$ over $\Rightarrow$. Begin the Natural Deduction proof with two hypotheses, $\forall x\ \Phi$ and $\forall x\ (\Phi \Rightarrow \Psi)$.

$$\dfrac{\dfrac{\dfrac{\forall x\ \Phi}{\Phi}\forall_E \qquad \dfrac{\forall x\ (\Phi \Rightarrow \Psi)}{\Phi \Rightarrow \Psi}\forall_E}{\Psi}\Rightarrow_E}{\forall x\ \Psi}\forall_I$$

Notice that the use of $\forall_I$ is acceptable, because x does not occur free in any of the hypotheses. (It does not matter that x occurs free in some of the intermediate steps.) Now, use $\Rightarrow_I$ twice—first with $\forall x\ \Phi$ and then with $\forall x(\Phi \Rightarrow \Psi)$ to obtain the theorem

$$\forall x\ (\Phi \Rightarrow \Psi) \Rightarrow (\forall x\ \Phi \Rightarrow \forall x\ \Psi).$$

Again, we would not expect to be able to prove the converse.

Example 5.8 Consider the (seemingly pointless) inference below.

$$\dfrac{\dfrac{\dfrac{\forall y\ \Phi}{\Phi}\forall_E}{\exists x\ \Phi}\exists_I}{\forall y\ \exists x\ \Phi}\forall_I$$

Because the variable x does not appear free in the conclusion, we may apply the $\exists_E$ rule and obtain

$$\dfrac{\exists x\ \forall y\ \Phi \quad \begin{array}{c}\cancel{\forall y\ \Phi}\\ \vdots\\ \forall y\ \exists x\ \Phi\end{array}}{\forall y\ \exists x\ \Phi}\exists_E$$

The theorem, after applying $\Rightarrow_I$, is

$$(\exists x\ \forall y\ \Phi) \Rightarrow (\forall y\ \exists x\ \Phi).$$

Notice that the implication holds in only one direction. If the quantifiers are the same, both directions can be proved. Using similar techniques, one can prove the theorems

$$(\forall x\, \forall y\, \Phi) \equiv (\forall y\, \forall x\, \Phi)$$

and

$$(\exists x\, \exists y\, \Phi) \equiv (\exists y\, \exists x\, \Phi).$$

Example 5.9 The interplay between negation and the quantifiers is interesting. Consider a simple inference with one undischarged hypothesis.

$$\dfrac{\dfrac{\neg\Phi \qquad \dfrac{\cancel{\forall x\, \Phi}}{\Phi}\,\forall_E}{\bot}\,\neg_E}{\neg\forall x\, \Phi}\,\neg_I$$

One can incorporate that inference into an application of the $\exists_E$ rule and obtain the theorem

$$(\exists x\, \neg\Phi) \Rightarrow (\neg\forall x\, \Phi).$$

The converse is a classical theorem, but not an intuitionistic one. The idea is that to prove $\exists x\, \neg\Phi$ intuitionistically, one must *exhibit* a value for x that makes $\neg\Phi$ true. It is not enough merely to disprove the assertion that all values of x make Φ true.

Using the classical rules, which essentially give us the equivalence of Ψ with $\neg\neg\Psi$, it is easy to prove the theorems

$$(\exists x\, \neg\Phi) \equiv (\neg\forall x\, \Phi)$$

and

$$(\forall x\, \neg\Phi) \equiv (\neg\exists x\, \Phi).$$

Example 5.10 Violating the conditions on the $\forall_I$ and $\exists_E$ rules can lead us to unsound conclusions. Without the condition on the latter, for example, we would have the inference

$$\dfrac{\dfrac{\dfrac{0 \doteq 0}{\exists x\, x \doteq 0}\,\exists_I \qquad \cancel{x \doteq 0}}{x \doteq 0}\,\exists_E\ \text{[Incorrect!]}}{\forall x\, x \doteq 0}\,\forall_I$$

Then, by using $\Rightarrow_I$ and discharging the assumption $0 \doteq 0$, we would obtain a proof of

$$0 \doteq 0 \Rightarrow \forall x\, x \doteq 0.$$

Surely, we do not want to accept this as a theorem of our system. The example is especially striking using $\doteq$, because we understand equality. But the example applies as well to *any* binary predicate. Using this "reasoning," we could prove that $S \times S$ is the only reflexive relation on a set S.

5.7 Sequent Calculus

The rules for the classical sequent calculus are augmented with four rules for quantifiers.

$$\frac{\Gamma, \Phi[x := t] \to \Delta}{\Gamma, \forall x\, \Phi \to \Delta}\forall_L \qquad \frac{\Gamma \to \Delta, \Phi[x := c]}{\Gamma \to \Delta, \forall x\, \Phi}\forall_R$$

$$\frac{\Gamma, \Phi[x := c] \to \Delta}{\Gamma, \exists x\, \Phi \to \Delta}\exists_L \qquad \frac{\Gamma \to \Delta, \Phi[x := t]}{\Gamma \to \Delta, \exists x\, \Phi}\exists_R$$

In the rules $\forall_L$ and $\exists_R$, the term t may be any term. In the rules $\exists_L$ and $\forall_R$, the constant symbol c may not appear (at all) in the lower sequent. It is very important to observe the condition on c!

Example 5.11 Here is a sequent calculus proof of the theorem $\exists x\, \Phi \Rightarrow \exists x\, \big(\Phi \vee \Psi\big)$.

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{}{\Phi[x := c] \to \Phi[x := c], \Psi[x := c]}\,\text{id}}{\Phi[x := c] \to \Phi[x := c] \vee \Psi[x := c]}\,\vee_R}{\Phi[x := c] \to \exists x\, \big(\Phi \vee \Psi\big)}\,\exists_R}{\exists x\, \Phi \to \exists x\, \big(\Phi \vee \Psi\big)}\,\exists_L}{\to \exists x\, \Phi \Rightarrow \exists x\, \big(\Phi \vee \Psi\big)}\,\Rightarrow_R$$

Reading from the top down, it is important that the use of $\exists_R$ come before the use of $\exists_L$, so that the restrictions on the latter can be obeyed.

Stopping the deduction at the penultimate step gives $\exists x\, \Phi \to \exists x\, \big(\Phi \vee \Psi\big)$. A similar deduction will show that $\exists x\, \Psi \to \exists x\, \big(\Phi \vee \Psi\big)$. Combining the two deductions with $\vee_L$ yields one direction of the distributive law for $\exists$ over $\vee$.

$$\cfrac{\cfrac{\exists x\, \Phi \to \exists x\, \big(\Phi \vee \Psi\big) \quad \exists x\, \Psi \to \exists x\, \big(\Phi \vee \Psi\big)}{\big(\exists x\, \Phi\big) \vee \big(\exists x\, \Psi\big) \to \exists x\, \big(\Phi \vee \Psi\big)}\,\vee_L}{\to \big(\exists x\, \Phi\big) \vee \big(\exists x\, \Psi\big) \to \exists x\, \big(\Phi \vee \Psi\big)}\,\Rightarrow_R$$

What goes wrong if you try to carry out the same derivation with $\wedge$ in place of $\vee$?

Example 5.12 The converse of the formula at the end of Example 5.11 is also a theorem. One begins with the following deduction.

$$\dfrac{\dfrac{\dfrac{}{\Phi[x := c] \rightarrow \Phi[x := c], \Psi[x := c]}\,\text{id}}{\Phi[x := c] \rightarrow \exists x\ \Phi, \exists x\ \Psi}\,\exists_R\text{, twice}}{\Phi[x := c] \rightarrow \big(\exists x\ \Phi\big) \vee \big(\exists x\ \Psi\big)}\,\vee_R$$

A similar deduction concludes with $\Psi[x := c] \rightarrow \big(\exists x\ \Phi\big) \vee \big(\exists x\ \Psi\big)$. One can then combine the two deductions.

$$\dfrac{\dfrac{\dfrac{\Phi[x := c] \rightarrow \big(\exists x\ \Phi\big) \vee \big(\exists x\ \Psi\big) \quad \Psi[x := c] \rightarrow \big(\exists x\ \Phi\big) \vee \big(\exists x\ \Psi\big)}{\Phi[x := c] \vee \Psi[x := c] \rightarrow \big(\exists x\ \Phi\big) \vee \big(\exists x\ \Psi\big)}\,\vee_L}{\exists x\ \big(\Phi \vee \Psi\big) \rightarrow \big(\exists x\ \Phi\big) \vee \big(\exists x\ \Psi\big)}\,\exists_L}{\rightarrow \exists x\ \big(\Phi \vee \Psi\big) \Rightarrow \big(\exists x\ \Phi\big) \vee \big(\exists x\ \Psi\big)}\,\Rightarrow_R$$

We therefore have the theorem $\exists x\ \big(\Phi \vee \Psi\big) \equiv \big(\exists x\ \Phi\big) \vee \big(\exists x\ \Psi\big)$.

What happens if you try to derive the similar formula with universal quantifiers instead of existential ones?

Example 5.13 Here is an intuitionistic derivation of $\exists x\ \neg\Phi \Rightarrow \neg\forall x\ \Phi$. We have simplified the presentation by omitting the last step, which uses $\Rightarrow_R$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Phi[x := c] \rightarrow \Phi[x := c]}\,\text{id}}{\neg\Phi[x := c], \Phi[x := c] \rightarrow \bot}\,\neg_L}{\neg\Phi[x := c], \forall x\ \Phi \rightarrow \bot}\,\forall_L}{\neg\Phi[x := c] \rightarrow \neg\forall x\ \Phi}\,\neg_R}{\exists x\ \neg\Phi \rightarrow \neg\forall x\ \Phi}\,\exists_L$$

Notice how $\bot$ was introduced on the right by $\neg_L$ and later removed by $\neg_R$. The rule $\neg_L$ will allow us to put *any* formula on the right, but it is common to use $\bot$ in cases like this.

Derivations like this one are relatively easy to construct, starting from the bottom and working upward. A good strategy is to apply the quantifier rules with conditions, $\forall_R$ and $\exists_L$, as close to the bottom of the derivation as possible.

Example 5.14 Here is a simple example that shows the importance of the condition on the use of $\forall_R$.

$$\dfrac{\dfrac{\overline{\underline{0} \doteq \underline{0} \to \underline{0} \doteq \underline{0}}\ \text{id}}{\underline{0} \doteq \underline{0} \to \forall x\, \underline{0} \doteq x}\ \forall_R,\ \text{incorrectly}}{\to \underline{0} \doteq \underline{0} \Rightarrow \forall x\, \underline{0} \doteq x}\ \Rightarrow_R$$

Using reasoning like this, one can "prove" the formula

$$\big(\forall x\, x \doteq x\big) \Rightarrow \big(\forall x\, \forall y\, x \doteq y\big).$$

Since the predicate symbol $\doteq$ is always interpreted as equality, the hypothesis of the formula is true in every model, but the conclusion is false in every model that has a domain with more than one element.

Notice that this example is very close to the example showing the importance of the analogous condition on the quantifier rules in the Natural Deduction system. Can you find an example that shows the importance of the condition on $\exists_L$?

5.8 Normal Forms

A formula is in **prenex form**, or **prenex normal form**, if it has all the quantifiers on the outside:

$$Q_1 x_1\ Q_1 x_1\ \ldots\ Q_1 x_1\ \Lambda$$

where each Q_i is either $\forall$ or $\exists$ and Λ is quantifier-free. Any formula can be transformed into an equivalent formula using the following equivalences, in which it is assumed that x does not occur as a free variable in Ψ.

$$\begin{aligned} \big(\forall x\ \Phi\big) \wedge \Psi &\equiv \forall x\ \big(\Phi \wedge \Psi\big) \\ \big(\forall x\ \Phi\big) \vee \Psi &\equiv \forall x\ \big(\Phi \vee \Psi\big) \\ \neg\forall x\ \Phi &\equiv \exists x\ \neg\Phi \\ \neg\exists x\ \Phi &\equiv \forall x\ \neg\Phi \end{aligned}$$

From these, one can derive other rules. Obvious examples are

$$\Psi \wedge \big(\forall x\ \Phi\big) \equiv \forall x\ \big(\Psi \wedge \Phi\big)$$

and

$$\big(\exists x\ \Phi\big) \wedge \Psi \equiv \exists x\ \big(\Phi \wedge \Psi\big).$$

Only slightly less obvious are the equivalences

$$\Psi \Rightarrow \big(\forall x\ \Phi\big) \equiv \forall x\ \big(\Psi \Rightarrow \Phi\big)$$

and

$$\big(\forall x\ \Phi\big) \Rightarrow \Psi \equiv \exists x\ \big(\Phi \Rightarrow \Psi\big).$$

If x does occur as a free variable in Ψ, then one can rename the bound variable x in $\forall x\ \Phi$. Choosing a new variable u that does not occur at all in $\big(\forall x\ \Phi\big) \wedge \Psi$, we have

$$\big(\forall x\ \Phi\big) \wedge \Psi \equiv \forall u\ \big(\Phi[x := u] \wedge \Psi\big).$$

Example 5.15 Consider the formula

$$\forall z\ \Big(p(x) \wedge \forall x\ \exists y\ \big(q(y,\ f(z)) \Rightarrow r(g(g(w, x),\ f(y)))\big)\Big).$$

Since x is free in $p(x)$, we rename the bound variable x as u.

$$\forall z\ \Big(p(x) \wedge \forall u\ \exists y\ \big(q(y,\ f(z)) \Rightarrow r(g(g(w, u),\ f(y)))\big)\Big).$$

We can then bring the quantifiers $\forall u$ and $\exists y$ out to the front and obtain

$$\forall z\ \forall u\ \exists y\ \Big(p(x) \wedge \big(q(y,\ f(z)) \Rightarrow r(g(g(w, u),\ f(y)))\big)\Big).$$

Taking some terminology from the propositional calculus, a **literal** is an atomic formula or the negation of an atomic formula. A **clause** is a disjunction of literals. A **CNF formula** is a conjunction of clauses. The methods from propositional calculus are sufficient to transform a quantifier-free formula into a prenex formula in which the quantifier-free part is in CNF.

Example 5.16 The quantifier-free part of the result in Example 5.15 is

$$p(x) \wedge \big(q(y,\ f(z)) \Rightarrow r(g(g(w, u),\ f(y)))\big).$$

It is equivalent to the CNF formula $\big(p(x)\big) \wedge \big(\neg q(y,\ f(z)) \vee r(g(g(w, u),\ f(y)))\big)$.

A formula is in **clausal form** if it is of the form

$$\forall x_1 \forall x_2 \ldots \forall x_k \Lambda$$

where Λ is a quantifier-free CNF formula. It is not true that any formula is equivalent to one in clausal form, but a weaker assertion can be proved.

Theorem 5.4 *Any formula Φ can be transformed into a clausal formula Ψ such that Φ is satisfiable if and only if Ψ is satisfiable.*

The transformation actually changes the underlying language by adding new constant and function symbols, called **Skolem** constants and functions. The first step in the transformation is to change Φ into an equivalent formula in prenex form. The second step is to put the quantifier-free part into CNF. Then the existential quantifiers are eliminated, one at a time, from left to right.

Suppose that the formula starts with an existential quantifier: $\exists y\, \Lambda$. Select a new constant symbol d, one that does not appear at all in the language, and replace the variable y with d. The resulting formula is $\Lambda[y := d]$. You should verify that $\exists y\, \Lambda$ is satisfiable if and only if $\Lambda[y := d]$ is satisfiable.

If the formula starts with a sequence of universal quantifiers,

$$\forall x_1 \forall x_2 \ldots \forall x_k \exists y\, \Lambda,$$

then we add a new function symbol. (The case in the previous paragraph is actually a special case, if we consider constants as 0-ary functions.) Let f be a k-ary function symbol that is not yet in the language, and replace the formula with

$$\forall x_1 \forall x_2 \ldots \forall x_k \Lambda[y := f(x_1, x_2, \ldots, x_k)].$$

Again, you should verify that the original formula is satisfiable if and only if the new one is satisfiable.

Example 5.17 From Examples 5.15 and 5.16, we see that the formula

$$\forall z \left(p(x) \wedge \forall x\, \exists y\, \left(q(y, f(z)) \Rightarrow r(g(g(w, x), f(y))) \right) \right)$$

is equivalent to

$$\forall z\, \forall u\, \exists y\, \left(\left(p(x) \right) \wedge \left(\neg q(y, f(z)) \vee r(g(g(w, u), f(y))) \right) \right). \qquad (2)$$

Let h be a new binary function symbol. Then formula (2) is satisfiable if and only if the clausal formula

$$\forall z\, \forall u\, \left(\left(p(x) \right) \wedge \left(\neg q(h(z, u), f(z)) \vee r(g(g(w, u), f(h(z, u)))) \right) \right)$$

is satisfiable.

5.9 Unification

A substitution σ is just a sequence of replacements of the form

$$[x_1 := t_1, x_2 := t_2, \ldots, x_k := t_k],$$

where the x_i's are variables and the t_i's are arbitrary terms. The substitutions are done in order, from left to right. The square brackets are reminiscent of our previous notation for substitution. The expression $E\sigma$, or equivalently $E[x_1 := t_1, \ldots, x_k := t_k]$, is the result of making the substitution in the expression E.

The **unification problem** asks when there is a substitution that can make two terms equal.

Example 5.18 Let t_1 be $g(x, z)$, let t_2 be $g(y, f(y))$, and let σ be the substitution $[y := x, z := f(x)]$. Then both $t_1\sigma$ and $t_2\sigma$ are equal to $g(x, f(x))$.

If t_1 and t_2 are terms and σ is a substitution satisfying $t_1\sigma = t_2\sigma$, then the terms t_1 and t_2 are **unifiable**. The substitution σ is a **unifier** of the two terms.

Example 5.19 In Example 5.18, the terms t_1 and t_2 are unifiable, and σ is a unifier. In contrast, the terms $g(x, x)$ and $g(y, f(y))$ are not unifiable.

A substitution σ is a **most general unifier** of terms t_1 and t_2 if each unifier τ of t_1 and t_2 can be written as $\tau = \sigma\upsilon$ for some substitution υ. That is to say, *any* unifier of t_1 and t_2 can be obtained from σ by a further substitution.

Unification has many uses. We will use it as a part of the resolution method for predicate logic. Unification is also the engine that drives the type inference mechanism of the programming language ML.

The Unification Algorithm is best stated in a general framework. Suppose that we have a sequence of equations

$$s_1 = t_1, s_2 = t_2, \ldots, s_k = t_k \tag{3}$$

and we seek a substitution σ that will make all of the equalities true:

$$s_1\sigma = t_1\sigma, s_2\sigma = t_2\sigma, \ldots, s_k\sigma = t_k\sigma.$$

The substitution σ can also be represented by a sequence of equations

$$x_1 = u_1, x_2 = u_2, \ldots, x_n = u_n, \tag{4}$$

where the variables x_i do not appear in the terms u_i. The Unification Algorithm will either transform the original equations (3) into a suitable substitution (4) or it will fail, indicating that there is no unifier. If there is a unifier, the Unification Algorithm will produce a most general unifier.

The Unification Algorithm iteratively applies the following steps to a sequence of equations. The algorithm is non-deterministic in that there is often a choice of which steps to apply. It terminates when there are no steps to apply or when it explicitly fails. In the former case, when there are no further steps to take, the resulting equations comprise a most general unifier.

0. If $s = t$ appears more than once among the equations, erase all but one instance of $s = t$.

1. Erase an equation of the form $x = x$, where x is a variable.

2. If x is a variable and t is a term that is not a variable, and $t = x$ appears among the equations, replace $t = x$ by $x = t$.

3. Suppose that $s = t$ appears among the equations and that neither s nor t is a variable.
   (a) If s and t are the *same* constant symbol, then erase the equation $s = t$.
   (b) If s is $f(u_1, \ldots, u_k)$ and t is $f(v_1, \ldots, v_k)$, then erase $s = t$ and add the equations $u_1 = v_1$ through $u_k = v_k$.
   (c) If s or t have any other forms, then fail immediately. This case applies when s and t are different constant symbols, when one of s and t is a constant symbol and the other is not, when s and t are constructed with different leading function symbols, and when s and t are constructed from function symbols of different arities.

4. If x is a variable, t is a term containing x, and $x = t$ appears among the equations, then fail immediately.

5. If x is a variable, t is a term *not* containing x, the equation $x = t$ appears, and x occurs in some of the other equations, then make the substitution $[x := t]$ in all the remaining equations.

Example 5.20 Consider the two equations

$$\begin{aligned} g(y) &= x \\ f(x, h(x), y) &= f(g(z), w, z) \end{aligned}$$

We can apply Step 2 to the first equation and Step 3(b) to the second, giving

$$\begin{aligned} x &= g(y) \\ x &= g(z) \\ h(x) &= w \\ y &= z \end{aligned}$$

One *could* apply Step 5 to the first equation and make a substitution for x, but it is easier to use the last equation. After applying Step 5 with $y = z$, eliminating the resulting duplicate, and applying Step 2, we obtain

$$\begin{aligned} x &= g(z) \\ w &= h(x) \\ y &= z \end{aligned}$$

Now apply Step 5 to the first equation to obtain a most general unifier.

$$\begin{aligned} x &= g(z) \\ w &= h(g(z)) \\ y &= z \end{aligned}$$

You should verify that the original equations are satisfied after these substitutions are made.

To apply Step 4 or 5, one must check the equation $x = t$ to see whether the variable x occurs in the term t. This is called the **occur check**. Some implementations of the Unification Algorithm omit the occur check for efficiency reasons, with the consequence that those implementations sometimes give wrong results.

Example 5.21 Let us see what happens when we apply the Unification Algorithm to the non-unifiable terms in Example 5.19. We begin with the single equation

$$g(x, x) = g(y, f(y)).$$

From Step 3(b), we obtain the two equations below.

$$\begin{aligned} x &= y \\ x &= f(y) \end{aligned}$$

There are only two possible steps to apply. If we apply Step 5 to the first equation, we obtain the equation $y = f(y)$ which fails the occur check. If we apply Step 5 to the second equation, we obtain the equation $f(y) = y$ which, after an application

of Step 2, also fails the occur check. No matter what we do, the algorithm will report (correctly) that the original terms are not unifiable.

Without the occur check, we would end up with the equations $x = f(y)$ and $y = f(y)$. An incorrect implementation could conclude that these equations constituted a most general unifier, or it could go on forever, generating longer and longer terms by repeatedly substituting $f(y)$ for y.

We do not prove the correctness of the unification algorithm here. See Theorem 4.3.3 in Ben Ari's book. Some steps of the proof are apparent. Without too much difficulty, one can prove for each step, that the equations are unifiable before the step if and only if they are unifiable after the step. Further, if the algorithm fails as a result of Step 3(c), then it is easy to see that the equations are not unifiable. It is a bit trickier to prove that the equations are not unifiable when the algorithm fails as a result of Step 4.

It is a bit difficult even to argue that the Unification Algorithm terminates. One cannot use induction on the number of equations, because Step 3(a) increases the number of equations. An alternative would be to use induction on the number of symbols in all the terms in the equations. Step 3(a) does reduce that number. But there are cases in which Step 5 will increase the total number of symbols. A combination of these ideas will succeed in showing that the algorithm terminates.

We show that each step can be applied only finitely many times. When Step 5 is applied to an equation $x = t$, then that equation will not be used again. Step 5 will be applied at most once for each variable. Next consider Step 3(b). At some stage in an execution of the algorithm, let F be the number of function symbols (counting repetitions) in all the equations. Step 3(b) reduces that number by 2, so there can be at most $F/2$ applications of Step 3(b) between two applications of Step 5. For the remaining steps, let E be the number of equations at any point in an execution of the algorithm. Step 2 can be applied only once to each equation, and Steps 0, 1, and 3(a) reduce the number of equations. Therefore, there can be at most $2E$ applications of Steps 0, 1, 2, and 3(a) between two applications of Step 3(b). Steps 3(c) and 4, of course, terminate the algorithm immediately.

The argument for the termination of the Unification Algorithm can be formalized by using the lexicographic ordering from the beginning of the course. Consider the set $\mathbb{N} \times \mathbb{N} \times \mathbb{N}$ of triples of natural numbers with the lexicographic ordering. At any stage in an execution of the algorithm, let V be the number of variables to which Step 5 has not been applied, F be the total number of function symbols in the equations, and E be the number of equations. The triple (V, F, E) is a measure

of how close to completion the algorithm is. Each step of the algorithm either reduces V, leaves V alone and reduces F, or leaves V and F alone and reduces E. The triple (V, F, E) decreases in the lexicographic ordering on each step. Since the lexicographic ordering is well-founded, there cannot be an infinite decreasing sequence of triples, and there cannot be an infinite execution of the algorithm.

The number of steps in an execution of the algorithm can be quite large, however. When V is decreased as a result of Step 5, the value of F may be "reset" to a much higher value. Similarly, when F is decreased as a result of Step 3(b), the number of equations E can increase.

Example 5.22 Here is a pathological example in which the most general unifier is exponentially long.

$$\begin{array}{rcl} x_1 & = & f(x_0, x_0) \\ x_2 & = & f(x_1, x_1) \\ & \vdots & \\ x_n & = & f(x_{n-1}, x_{n-1}) \end{array}$$

With n applications of Step 5, working from top to bottom, the Unification Algorithm produces an equation $x_n = t$, where t is a long term containing only x_0 and f. In t, there are $2^n - 1$ occurrences of the function symbol f. A naive implementation (like ours) of the Unification Algorithm will take exponential time because the occur check must be applied to exponentially long formulas. To see the need for the occur check, consider the case where the first equation is $x_1 = f(x_n, x_n)$.

There are many approaches to solving the unification problem. Because it is used so frequently in so many places, computer scientists have studied it carefully and come up with creative optimizations. The version of the Unification Algorithm presented here is easy to understand but is not particularly fast. There are algorithms which solve the unification problem in linear time, in part by taking advantage of sophisticated representations of substitutions. Example 5.22 shows that they cannot "write out" the substitution explicitly in linear time.

5.10 Resolution

In the Resolution Procedure for propositional logic, we operated on clauses which contained clashing literals. There, a pair of clashing literals consisted of a proposition letter and its negation. In predicate logic, the literals are atomic formulas and their negations. But it is too restrictive to consider a clashing pair to be an atomic formula and its negation. Instead, a clashing pair of literals consists of two

atomic formulas, $p(t_1, t_2, \ldots, t_k)$ and $\neg p(u_1, u_2, \ldots, u_k)$, in which the terms t_i are unifiable with the terms u_i.

An atomic formula and its negation are **complements** of one another. We state the Resolution Rule for predicate logic as follows:

> Let C_1 and C_2 be clauses with no variables in common. Suppose that ℓ_1 is an element of C_1, ℓ_2 is an element of C_2, and σ is a substitution such that $\ell_1\sigma$ and $\ell_2\sigma$ are complements of one another. The **resolvent** of the clauses C_1 and C_2 is
>
> $$\mathrm{Res}(C_1, C_2) = \big(C_1\sigma \setminus \{\ell_1\sigma\}\big) \cup \big(C_2\sigma \setminus \{\ell_2\sigma\}\big).$$

Example 5.23 Perhaps surprisingly, a resolvent can have significantly fewer elements than the original clauses. Consider the case where C_1 is $\{\neg p(c, c, c)\}$ and C_2 is $\{p(x, y, z), p(y, z, x), p(z, x, y)\}$. No matter which literal in C_2 one chooses, the unifier σ is a substitution that replaces all three variables with the constant symbol c, and $C_2\sigma$ has only one element. The resolvent is the empty clause $\square$.

If $x_1, x_2, \ldots, x_k$ are the variables appearing in a clause C, then the **universal closure** of C is $\forall x_1 \forall x_2 \ldots \forall x_k C$. In the context of resolution, when we say that a clause is satisfiable, we mean that its universal closure is satisfiable. Similarly, to say that a set of clauses is satisfiable means that the universal closures of all the clauses are mutually satisfiable.

Because clauses are assumed to be universally quantified, any variable in a clause is bound to a universal quantifier. We may therefore replace all the occurrences of the variable x in one clause with a new variable without changing the intended meaning of the clause. Thus, we can always satisfy the restriction in the Resolution Rule that the two clauses do not share variables.

The Resolution Procedure for predicate logic is the same as the procedure for propositional logic; the only change is that we use the new version of the Resolution Rule. Starting with a set S of clauses, we let $S_0 = S$ and let S_{k+1} be the result of applying the resolution rule to S_k. The process terminates in one of two ways.

- If the resolvent is ever the empty clause $\square$, then declare the set S to be unsatisfiable and terminate.
- If there is no way to choose clashing clauses so that S_{k+1} is different from S_k, then declare the set S to be satisfiable and terminate.

Recall that, in the first case, we say S is **refuted.** The Resolution Procedure is a way of checking for satisfiability, but it is often used to show that a formula is valid by refuting the negation of the formula.

Unlike the Resolution Procedure for propositional logic, there is no guarantee that the process halts. We shall see an example shortly.

Example 5.24 The Resolution Procedure verifies that the formula $p(a) \Rightarrow \exists x\ p(x)$ is valid. Its negation is equivalent to the clausal formula $p(a) \wedge \forall x\, \neg p(x)$. Apply the Resolution Procedure to the two clauses $p(a)$ and $\neg p(x)$. There is no choice of literals, because there is only one literal in each clause. The literals clash under the substitution $[x := a]$. The Resolution Rule gives $\Box$ as the resolvent, so the clausal formula is not satisfiable, and its negation, $p(a) \Rightarrow \exists x\ p(x)$, is valid.

Example 5.25 The formula $\forall x\, \exists y\, \big(p(x, y)\big) \Rightarrow \exists y\, \forall x\, \big(p(x, y)\big)$ is not valid, so the Resolution Procedure should show that the negation of the formula is satisfiable. The negation is

$$\forall x\, \exists y\, \big(p(x, y)\big) \wedge \forall y\, \exists x\, \big(\neg p(x, y)\big). \tag{5}$$

Introducing a Skolem functions f and g to get rid of the existential quantifiers, we obtain the clauses $p(x, f(x))$ and $\neg p(g(y), y$.

An attempt to unify the terms fails with an occur check, so there is no way to unify the two literals, and the Resolution Procedure stops immediately without producing the empty clause, so formula (5) is satisfiable.

Notice that we are taking the conclusions of the Resolution Procedure on faith. We have not yet explained *why* a formula is satisfiable if the procedure halts without producing the empty clause.

Example 5.26 Consider a language with three constant symbols ϵ, a, and b and one binary function symbol $\hat{}$. Form the following axioms:

$$\begin{array}{ll} \text{A1:} & \forall x\, \big(\epsilon \,\hat{}\, x \doteq x\big) \\ \text{A2a:} & \forall x\, \forall y\, \Big(\big(x \doteq y\big) \equiv \big(x \,\hat{}\, \mathsf{a} \doteq y \,\hat{}\, \mathsf{a}\big)\Big) \\ \text{A2b:} & \forall x\, \forall y\, \Big(\big(x \doteq y\big) \equiv \big(x \,\hat{}\, \mathsf{b} \doteq y \,\hat{}\, \mathsf{b}\big)\Big) \end{array}$$

The intended interpretation is that the variables range over strings in the alphabet $\{\mathsf{a}, \mathsf{b}\}$. The constant symbol ϵ denotes the empty string, and a and b denote the one-element strings. The function symbol $\hat{}$ denotes string concatenation. There are, of course, many other interpretations.

We might ask if the formula $\epsilon\,\hat{}\,\epsilon \doteq \epsilon$ is a consequence of the axioms. It is in fact a consequence of A1, and we can refute (in one step) A1 and the negation of the desired consequence.

$$\epsilon\,\hat{}\,x \doteq x$$
$$\neg\big(\epsilon\,\hat{}\,\epsilon \doteq \epsilon\big)$$

One can easily extend the refutation to show that $\epsilon\,\hat{}\,t \doteq t$ is a consequence of axiom A1, for all terms t. In particular, $\epsilon\,\hat{}\,(\epsilon\,\hat{}\,\epsilon) \doteq \epsilon\,\hat{}\,\epsilon$.

Example 5.27 All consequences of the axioms are facts about strings, but not all facts about strings are consequences of the axioms. For example, we cannot take Example 5.26 one step further and conclude that $\epsilon\,\hat{}\,(\epsilon\,\hat{}\,\epsilon) \doteq \epsilon$. We *can* do it, however, with an additional axiom, the transitivity axiom for $\doteq$.

$$\epsilon\,\hat{}\,x \doteq x$$
$$\neg\big(x \doteq y\big) \vee \neg\big(y \doteq z\big) \vee \big(x \doteq z\big)$$
$$\neg\big(\epsilon\,\hat{}\,(\epsilon\,\hat{}\,\epsilon) \doteq \epsilon\big)$$

The literal in the last clause clashes with the final literal in the second clause under the substitution $[x := \epsilon\,\hat{}\,(\epsilon\,\hat{}\,\epsilon), z := \epsilon]$. The resolvent is

$$\neg\big(\epsilon\,\hat{}\,(\epsilon\,\hat{}\,\epsilon) \doteq y\big) \vee \neg\big(y \doteq \epsilon\big).$$

The first clause clashes with the axiom under the substitution $[x := \epsilon\hat{}\epsilon,\ y := \epsilon\hat{}\epsilon]$. The resolvent this time is

$$\neg\big(\epsilon\,\hat{}\,\epsilon \doteq \epsilon\big),$$

which, as we saw in Example 5.26, clashes with the axiom and produces the empty clause as the resolvent.

Example 5.28 Continuing with the axiom system of Example 5.26, is

$$\forall x\ \big(x\,\hat{}\,\epsilon \doteq x\big)$$

a consequence of the axioms? Let us try a special case, asking if $\mathsf{a}\,\hat{}\,\epsilon \doteq \mathsf{a}$ is a consequence. Add the negation of the formula to the axioms, change everything to clausal form, and try to refute these clauses:

$$\begin{aligned}
&x \doteq \epsilon\,\hat{}\,x \\
&\neg\big(x \doteq y\big) \vee \big(x\,\hat{}\,\mathsf{a} \doteq y\,\hat{}\,\mathsf{a}\big) \\
&\neg\big(x\,\hat{}\,\mathsf{a} \doteq y\,\hat{}\,\mathsf{a}\big) \vee \big(x \doteq y\big) \\
&\neg\big(x \doteq y\big) \vee \big(x\,\hat{}\,\mathsf{b} \doteq y\,\hat{}\,\mathsf{b}\big) \\
&\neg\big(x\,\hat{}\,\mathsf{b} \doteq y\,\hat{}\,\mathsf{b}\big) \vee \big(x \doteq y\big) \\
&\neg\big(\mathsf{a}\,\hat{}\,\epsilon \doteq \mathsf{a}\big)
\end{aligned} \qquad (6)$$

All clauses except the last clause are true in the string model, so we it will do us no good to look for clashes among them. The only "useful" clashes are between $\neg(\mathsf{a} \mathbin{\hat{}} \epsilon \doteq \mathsf{a})$ and the non-negated equalities.

The formula $\neg(\mathsf{a} \mathbin{\hat{}} \epsilon \doteq \mathsf{a})$ does not unify with $x \doteq \epsilon \mathbin{\hat{}} x$ or $x \mathbin{\hat{}} \mathsf{a} \doteq y \mathbin{\hat{}} \mathsf{a}$ or $x \mathbin{\hat{}} \mathsf{b} \doteq y \mathbin{\hat{}} \mathsf{b}$; it only unifies with the two instances of $x \doteq y$. The resolvents in those two cases are

$$\neg\big((\mathsf{a} \mathbin{\hat{}} \epsilon) \mathbin{\hat{}} \mathsf{a} \doteq \mathsf{a} \mathbin{\hat{}} \mathsf{a}\big) \text{ and } \neg\big((\mathsf{a} \mathbin{\hat{}} \epsilon) \mathbin{\hat{}} \mathsf{b} \doteq \mathsf{a} \mathbin{\hat{}} \mathsf{b}\big).$$

We can continue with resolvents of the same form, adding more a's and b's on the right, but those steps do not get us closer to a refutation. It appears that the Resolution Procedure will go on forever.

As presented here, the Resolution Procedure is not a *decision* procedure for predicate logic. The procedure satisfies three desirable properties.

1. If the Resolution Procedure produces the empty clause $\square$ from a set S of clauses, then the set S is unsatisfiable.
2. If a set S of clauses is unsatisfiable, then the Resolution Procedure will produce the empty clause $\square$.
3. If the Resolution Procedure, applied to a set S of clauses, halts without producing the empty clause $\square$, then the set S is satisfiable.

As shown in Example 5.28, the fourth "desirable property" is not as strong as we would like.

4. If S is a satisfiable set of clauses *and the Resolution Procedure comes to a halt,* then the Resolution Procedure will not produce the empty clause $\square$.

There are cases, like the set of clauses in (6), for which the search for $\square$ will continue indefinitely.

There is, however, an interpretation that will satisfy the clauses in (6). One can take the domain of the intended interpretation, strings over a two-element alphabet, and the corresponding interpretation of the constant symbols. Interpret the function $\hat{}$ by setting the value of $x \mathbin{\hat{}} y$ is the usual string concatenation when y is not the empty string. When y is the empty string, set

$$x \mathbin{\hat{}} \epsilon = \begin{cases} \epsilon & \text{if } x = \epsilon, \\ x \mathbin{\hat{}} \mathsf{b} & \text{if } x \text{ ends in } \mathsf{a}\text{, and} \\ x \mathbin{\hat{}} \mathsf{a} & \text{if } x \text{ ends in } \mathsf{b}. \end{cases}$$

Example 5.29 If we want to use the Resolution Procedure to conclude $\forall x\ \big(x \mathbin{\hat{}} \epsilon \doteq x\big)$ from some set of axioms, it will not help just to add axioms for equality. A refutation is possible if we add the equality axioms *and* the associative property of string concatenation. You are invited to show that the following set is refutable.

$$
\begin{array}{l}
\epsilon \mathbin{\hat{}} x \doteq x \\
\neg\big(x \doteq y\big) \vee \big(x \mathbin{\hat{}} \mathsf{a} \doteq y \mathbin{\hat{}} \mathsf{a}\big) \\
\neg\big(x \mathbin{\hat{}} \mathsf{a} \doteq y \mathbin{\hat{}} \mathsf{a}\big) \vee \big(x \doteq y\big) \\
\neg\big(x \doteq y\big) \vee \big(x \mathbin{\hat{}} \mathsf{b} \doteq y \mathbin{\hat{}} \mathsf{b}\big) \\
\neg\big(x \mathbin{\hat{}} \mathsf{b} \doteq y \mathbin{\hat{}} \mathsf{b}\big) \vee \big(x \doteq y\big) \\
(x \mathbin{\hat{}} y) \mathbin{\hat{}} z \doteq x \mathbin{\hat{}} (y \mathbin{\hat{}} z)
\end{array}
$$

$$
\begin{array}{l}
x \doteq x \\
\neg\big(x \doteq y\big) \vee \big(y \doteq x\big) \\
\neg\big(x \doteq y\big) \vee \neg\big(y \doteq z\big) \vee \big(x \doteq z\big)
\end{array}
$$

$$\neg\big(d \mathbin{\hat{}} \epsilon \doteq d\big)$$

Notice that we added a Skolem constant d to eliminate the existential quantifier that appeared with the negation of $\forall x\ \big(x \doteq \epsilon \mathbin{\hat{}} x\big)$.

6 Logic and Programming

6.1 Hoare Logic

One approach to program correctness was developed by C.A.R. Hoare. It is one of several formulations that uses ideas of axiomatization from logic.

Start with any language for predicate logic. Using the same variables, design a simple programming language as follows.

- The variables in the programming language are the same as the variables in the logic language.
- The expressions in the programming language are the terms of the logic language.
- There are in the programming language assignment statements of the form `x=t;` in which `x` is a variable and `t` is an expression.

- There is in the programming language an empty statement, denoted `;`.
- The programming language permits sequencing: The construction `S;T;` is interpreted as first executing the statement `S;` and then `T;`.
- The programming language permits constructions of the form

  ```
  if (c) then S; else T;
  ```

  where `c` is a boolean-valued expression without side-effects.
- The programming language permits constructions of the form

  ```
  while (c) S;
  ```

  where, again, `c` is a boolean-valued expression without side-effects.

A **Hoare triple** is a string $\{p\}$ `S;` $\{q\}$ in which p and q are formulas in the logic and `S;` is a statement in the programming language. The intended interpretation is that if p is true before executing the statement `S;`, then q will be true afterward. The condition p is called the **precondition**, and q is called the **postcondition**.

We work in a system in which we wish to derive both logical formulas and Hoare triples. We choose a system of deduction (like the Sequent Calculus) for the logic and apply these rules of inference.

0. $\dfrac{}{p}$ for any logical formula p provable in the underlying logical system.

1. $\dfrac{}{\{p\}\ \texttt{;}\ \{p\}}$

2. $\dfrac{\{p\}\ \texttt{S;}\ \{q\},\quad \{q\}\ \texttt{T;}\ \{r\}}{\{p\}\ \texttt{S;T;}\ \{r\}}$

3. $\dfrac{\{p \wedge c\}\ \texttt{S;}\ \{q\},\quad \{p \wedge \neg c\}\ \texttt{T;}\ \{q\}}{\{p\}\ \texttt{if (c) then S; else T;}\ \{q\}}$

4. $\dfrac{\{p \wedge c\}\ \texttt{S;}\ \{p\}}{\{p\}\ \texttt{while (c) S;}\ \{p \wedge \neg c\}}$

5. $\dfrac{}{\{p[x := t]\}\ \texttt{x=t;}\ \{p\}}$

6. $\dfrac{p \Rightarrow p',\quad \{p'\}\ \texttt{S;}\ \{q'\},\quad q' \Rightarrow q}{\{p\}\ \texttt{S;}\ \{q\}}$

Notice that rules 1 through 4 correspond to the constructs of the programming language. For obvious reasons, the condition p in the while rule 4 is called a **loop invariant.**

In the next several examples, we work through the details of analyzing the following fragment that implements binary search through an ordered array `ary` with `N` elements. We desire, when the code has finished executing, that the variable `low` contain the least index of an array element that is not less than `target`.

```
low = 0;
high = N;
while (low < high) {
  mid = (low + high)/2;
  if (ary[mid] < target)
     then low = mid+1;
     else high = mid;
}
```

Example 6.1 Assignment rule 5 immediately gives the Hoare triple

$$\{\texttt{low} \leq \texttt{mid}\}\ \texttt{high=mid;}\ \{\texttt{low} \leq \texttt{high}\}.$$

Example 6.2 The triple

$$\{\texttt{low} \leq \texttt{mid} < \texttt{high}\}\ \texttt{high=mid;}\ \{\texttt{low} \leq \texttt{high}\}$$

is derived from the result of Example 6.1 and the consequence rule 6.

Example 6.3 Assuming that we are dealing with integers, $\texttt{mid} < \texttt{high}$ implies that $\texttt{mid} + 1 \leq \texttt{high}$. We can then make a deduction similar to the previous examples:

$\{\texttt{low} \leq \texttt{mid} < \texttt{high}\}$
$\{\texttt{mid} + 1 \leq \texttt{high}\}$
`low=mid+1;`
$\{\texttt{low} \leq \texttt{high}\}$

Example 6.4 From Example 6.2 and the consequence rule 6, we have

$\{c \wedge (\texttt{low} \leq \texttt{mid} < \texttt{high})\}$
`high=mid;`
$\{\texttt{low} \leq \texttt{high}\}$

From Example 6.3 and the consequence rule again, we have

$\{\neg c \wedge (\texttt{low} \leq \texttt{mid} < \texttt{high})\}$
```
low=mid+1;
```
$\{\texttt{low} \leq \texttt{high}\}$

These hold for any condition c, so in particular they hold for the condition in the `if` statement of binary search. We conclude from the conditional rule 3 that

$\{\texttt{low} \leq \texttt{mid} < \texttt{high}\}$
```
if (ary[mid] < target)
  then low=mid+1;
  else high=mid;
```
$\{\texttt{low} \leq \texttt{high}\}$

Example 6.5 We next look at a larger block of code and combine some of the previous results. From $\texttt{low} < \texttt{high}$, we may conclude that

$$\texttt{low} < (\texttt{low} + \texttt{high})/2 < \texttt{high}$$

when working over floating point numbers. Over the integers, there may be truncation upon division, and we have

$$\texttt{low} \leq (\texttt{low} + \texttt{high})/2 < \texttt{high}$$

From the consequence rule and the assignment rule, we obtain the Hoare triple

$\{\texttt{low} < \texttt{high}\}$
```
mid = (low+high)/2;
```
$\{\texttt{low} \leq \texttt{mid} < \texttt{high}\}$

Using the sequence rule 2 to combine this triple with the one from Example 6.4, we obtain

$\{\texttt{low} < \texttt{high}\}$
```
mid = (low+high)/2;
if (ary[mid] < target)
  then low=mid+1;
  else high=mid;
```
$\{\texttt{low} \leq \texttt{high}\}$

Finally, using the while rule 4, we obtain a Hoare triple for the loop of binary search.

$\{$low $\leq$ high$\}$
```
while (low < high) {
  mid = (low+high)/2;
  if (ary[mid] < target)
    then low=mid+1;
    else high=mid;
}
```
$\{(\texttt{low} \leq \texttt{high}) \wedge \neg(\texttt{low} < \texttt{high})\}$

From the postcondition, we may conclude that $\texttt{low} = \texttt{high}$.

Example 6.6 Let us consider more interesting conditions on the code for binary search. How can we complete the precondition of the following Hoare triple?

$\{\ \text{Pre0} \wedge \texttt{ary[mid]} < \texttt{target}\}$
```
low = mid+1;
```
$\{\forall i\ (0 \leq i < \texttt{low} \Rightarrow \texttt{ary}[i] < \texttt{target})\}$

Working from the bottom, we can make some progress with the assignment rule 5.

$\{\ \text{Pre0} \wedge \texttt{ary[mid]} < \texttt{target}\}$
$\{\forall i\ (0 \leq i \leq \texttt{mid} \Rightarrow \texttt{ary}[i] < \texttt{target})\}$
$\{\forall i\ (0 \leq i < \texttt{mid} + 1 \Rightarrow \texttt{ary}[i] < \texttt{target})\}$
```
low = mid+1;
```
$\{\forall i\ (0 \leq i < \texttt{low} \Rightarrow \texttt{ary}[i] < \texttt{target})\}$

To get from the first line to the second, we must know that the array elements are ordered and that the order relation is transitive. To get from the second line to the third, we must know something about the relationship between addition and the usual ordering on the integers. Usually, the facts about orderings and integers are left unstated and we take Pre0 to be a statement about the ordering of elements in ary:

$$\forall i\, \forall j\ (0 \leq i < j < N \Rightarrow \texttt{ary}[i] \leq \texttt{ary}[j])$$

If we wanted to be more complete, we could take $p(\ ,\ )$ to be a predicate symbol intended to represent the (strict) ordering on integers, $q(\ ,\ )$ to be a predicate symbol for the ordering on array elements, $a(\ )$ to be a function symbol for array accesses, and $s(\ )$ to be a function symbol the arithmetic successor function. Then Pre0 might consist of the conjunction of the following nine assertions.

$$
\begin{array}{ll}
\text{I0} & \forall i\,\forall j\ \big(\big(0 \doteq i \vee p(0,i)\big) \wedge p(i,j) \wedge p(j,N) \Rightarrow q(a(i),a(j))\big) \\
\\
\text{J0} & \forall i\,\forall j\ \big(p(i,j) \Rightarrow \neg i \doteq j\big) \\
\text{J1} & \forall i\,\forall j\ \big(p(i,j) \vee i \doteq j \vee p(j,i)\big) \\
\text{J2} & \forall i\,\forall j\,\forall k\ (p(i,j) \wedge p(j,k) \Rightarrow p(i,k)) \\
\text{J3} & \forall i\,\forall j\ \big(p(i,s(j)) \Rightarrow p(i,j) \vee i \doteq j\big) \\
\text{J4} & \big(0 \doteq \texttt{mid} \vee p(0,\texttt{mid})\big) \wedge p(\texttt{mid},N) \\
\\
\text{K0} & \forall x\,\forall y\ \big(p(x,y) \Rightarrow \neg x \doteq y\big) \\
\text{K1} & \forall x\,\forall y\ \big(p(x,y) \vee x \doteq y \vee p(y,x)\big) \\
\text{K2} & \forall x\,\forall y\,\forall z\ (p(x,y) \wedge p(y,z) \Rightarrow p(x,z))
\end{array}
$$

One could then prove, in any predicate logic system, that

$$
\begin{array}{l}
\text{Pre0} \wedge q(a(\texttt{mid}),\texttt{target}) \Rightarrow \\
\quad \forall i\ \big(\big(0 \doteq i \vee p(0,i)\big) \wedge p(i,s(\texttt{mid})) \Rightarrow q(a(i),\texttt{target})\big).
\end{array}
$$

Notice, in J4, that it is necessary to state explicitly facts like $0 \leq \texttt{mid} < N$. Most humans would overlook these obvious assertions, but such "boundary cases" turn out to be a source of common errors. Properly applied, Hoare logic can help to avoid such errors.

Example 6.7 Let us now step back and consider the situation more broadly. Assume that `ary` is an array of integers indexed from 0 through $N-1$. Consider the following assertions:

$$
\begin{array}{ll}
\text{I0:} & \forall i\,\forall j\ (0 \leq i < j < N \Rightarrow \texttt{ary}[i] \leq \texttt{ary}[j]) \\
\text{I1:} & \texttt{low} \leq \texttt{high} \\
\text{I2:} & \forall i\ (0 \leq i < \texttt{low} \Rightarrow \texttt{ary}[i] < \texttt{target}) \\
\text{I3:} & \forall i\ (\texttt{high} \leq i < N \Rightarrow \texttt{target} \leq \texttt{ary}[i])
\end{array}
$$

Let Inv be the conjunction of I0 through I3. Then we can establish the following Hoare triples.

{ I0 }
`low=0;`
`high=`N`;`
{ Inv }

and

```
{ Inv }
while (low < high) {
  mid = (low+high)/2;
  if (ary[mid] < target)
    then low=mid+1;
    else high=mid;
}
{ Inv ∧ high ≤ low }
```

From the postcondition, we may deduce that $\texttt{low} = \texttt{high}$ and

$$\forall i\, \big((0 \le i < \texttt{low} \Rightarrow \texttt{ary}[i] < \texttt{target}) \wedge \\ (\texttt{high} \le i < N \Rightarrow \texttt{target} \le \texttt{ary}[i])\big).$$

If `target` appears anywhere in the array, then it appears first at index `low`.

A program fragment is **annotated** if *all* the statements have pre- and postconditions. Obviously, a postcondition to one statement is a precondition to the next statement. Sometimes, there are several conditions in a row, with each one following from the previous one. These sequences of conditions are to make the reasoning clearer. A annotated version of binary search appears in Table 6. A correctly annotated program may be considered to be a **theorem** of Hoare logic.

6.2 Termination

The characterization of Hoare triples in the previous section was inexact. It said that the interpretation of the triple $\{p\}$ `S;` $\{q\}$ means "if p is true before `S;` is executed, then q is true afterward." The correct meaning is "if p is true beforehand *and the statement* `S;` *terminates,* then q is true afterward."

If all we wanted were invariants, the result of Example 6.7 could easily have been

```
{0 < N}
low=0;
high=N;

while (low < high)
  do nothing;
{ Inv ∧ low = high }
```

A program is **partially correct** if, whenever it gives an answer, that answer is correct. A program is **totally correct** if it always gives a correct answer. We shall see in the next section why it is necessary to consider termination separately.

```
{ I0 }
low=0;
high=N;
{ Inv }
while (low < high) {
  { Inv ∧ low < high}
  { Inv ∧ low ≤ (low + high)/2 < high}
  mid = (low+high)/2;
  { Inv ∧ low ≤ mid < high}
  if (ary[mid] < target)
    then { Inv ∧ low ≤ mid < high ∧ ary[mid] < target}
         low = mid+1;
         { Inv }
    else { Inv ∧ low ≤ mid < high ∧ target < ary[mid]}
         high = mid;
         { Inv }
  { Inv }
}
{ Inv ∧ high ≤ low }
{low = high ∧
 ∀i ((0 ≤ i < low ⇒ ary[i] < target) ∧
    (high ≤ i < N ⇒ target ≤ ary[i]))}
```

Table 6: An (almost completely) annotated version of binary search.

Often, one can verify termination by considering an **termination function,** an expression constructed from program variables. If the function takes on only non-negative integral values, and if the value of the function decreases after each iteration of the loop, then the loop will terminate.

For binary search, the expression `high - low` is a termination function. Given the invariants of Example 6.7, it is easy to show that the following Hoare triple is valid.

$0 <$ `high - low` $= T$

```
mid = (low+high)/2;
if (ary[mid] < target)
  then low=mid+1;
  else high=mid;
```

$0 \leq$ `high - low` $< T$

The integer value `high - low` cannot decrease indefinitely and still remain positive. There must be a point at which `high - low` becomes zero and the loop terminates.

For most loops, it is obvious that the body of the loop "makes progress" toward a solution. The termination function simply quantifies that property, and it is often obvious what the termination function should be. Not surprisingly, the termination function is frequently closely related to the termination condition for the loop.

Although we cannot speak of recursion in our demonstration language (because it does not have methods or procedures), the same ideas apply to showing that a recursive program terminates. One simply has to find a termination function T on the parameters of a program which satisfies two properties:

1. the values of T are non-negative integers, and
2. the values of T for a recursive call are *smaller* than the values for the program.

This method is a formalization of the natural practice of ensuring that a recursive call is done on a *simpler case* of the problem.

The techniques described here are easy and natural, and they apply in a variety of common situations. It is not surprising, perhaps, that they do not apply in all cases. A stronger conclusion is that it is in general *impossible* to prove termination in all cases.

6.3 The Halting Problem

The simple programming language of the previous sections is sufficient to express all programs. Thus, any result about termination of programs in our simple language applies to termination of programs in any language. Moreover, our simple language is sufficient to express any algorithm that could (more clearly and more elegantly) be expressed in another programming language.

Although we can think of input as coming from an initialization section of our code, it is clearer for our informal presentation to imagine that the language does have some input primitives. It is also clearer to consider a language that includes functions and recursion.

A program is simply a string of characters, which in turn can be reduced to a string of bits. Thus, a program can be the input to another program. Is there a program H which takes as input a program p and a string i and always halts with the following result?

- If p halts with input i, then $H(p, i)$ terminates with the result 1 in the variable `x`.
- If p does not halt with input i, then $H(p, i)$ terminates with the result 0 in the variable `x`.

The question is widely known as "The Halting Problem," and the answer is "no." There is no such "halting program H."

Suppose, for contradiction, that H is a halting program, and let D be the program that takes one program p, runs $H(p, p)$, and then executes the loop with empty body `while (0 < x) ;`

What happens when D is applied to itself?

- If $D(D)$ halts, then $H(D, D)$ must have halted with `x` $= 0$. But by the characterization of H, that means that $D(D)$ does not halt—a contradiction.
- If $D(D)$ does not halt, then $H(D, D)$ must have halted with `x` $= 1$. Again, by the characterization of H, that means that $D(D)$ does halt—again a contradiction.

Either way, we have a contradiction, and we conclude that there is no such H.

Stated more carefully, the conclusion is that, given any algorithm H, there is an example for which H gives the "wrong answer" to the halting question. Therefore,

no algorithm can correctly answer all questions about halting. Since there is no way to tell, in general, if a given program halts, there is no completely general method to prove termination. Thus termination is "special" and we separate termination proofs from other correctness proofs.

Another conclusion is that there is no way to determine if a particular section of a program is ever executed. If we could answer that question, we could answer questions about halting by just asking if the last statement of a program was executed.

In closing, we observe that the sketches given in this section are not complete proofs. To be rigorous, we would have to show how a program can take another program as input, modify it, and simulate the execution of the modification. The topics of this section and the next are covered much more deeply and completely in the Theory of Computation course.

6.4 Incompleteness in Logic

The reasoning of the last section can be applied to logical deduction. Let us fix a system of deduction, say the sequent calculus. It is easy, in principle, to check that a deduction is correct. We just look at each step and see that the rules are followed. The process can be done mechanically.

There are a few simple cases in which a similar mechanical process will *generate* a deduction of a given formula. When we considered the classical sequent calculus for propositional logic, we saw that we can algorithmically apply the rules and construct a deduction of a formula if one exists. If there is no deduction, our algorithmic process fails explicitly.

The happy situation that occurs in the classical propositional sequent calculus is rare. For most logics, we cannot reliably generate a deduction or refutation for an arbitrary formula. If we have a logical system that is sufficiently rich, a self-referential argument like the one in the last section can be used to prove that there is no algorithm that will tell us if a formula has a derivation. Further, there is a formula that is neither derivable nor refutable. There is some question that our "sufficiently rich" system cannot answer!

Roughly speaking, a "sufficiently rich" system is one in which we can in the object language manipulate symbols and make definitions by induction. Since we can encode strings of symbols (in a logical system or a programming language, say) as integers, it is enough to have the integers and their common properties. As in the last section, the presentation here is a sketch and not a complete proof. We

are particularly sloppy in bluring the distinction between integers and terms which represent them.

With an adequate encoding of the language, we may define a formula WFF( ) with one free variable such that WFF(n) is provable for a natural number n if and only if n is the encoding of a well-formed formula. In the same spirit, we can define formulas for terms, substitution, and derivations. In particular, we can define a formula Bew(,) of three free variables such that

> $\mathrm{Bew}(k, m, n)$ is provable for natural numbers k, m, and n if and only if m is the encoding for a formula Φ with the free variable x and k is the encoding for a correct derivation of $\Phi[x := n]$.

Now let $\mathcal{D}(x)$ be the formula $\forall z\ (\neg\mathrm{Bew}(z, x, x))$, and let d be the integer which encodes it. Consider the formula $\mathcal{D}(d)$; it is

$$\forall z\ (\neg\mathrm{Bew}(z, d, d)), \tag{1}$$

which asserts that there is no encoding for a derivation of the formula $\mathcal{D}(d)$. Therefore, $\mathcal{D}(d)$ is a formula which asserts its own unprovability!

More precisely, if $\mathcal{D}(d)$ has a derivation, then formula (1) is a theorem. But if $\mathcal{D}(d)$ has a derivation, we can let e be an integer that encodes the derivation. By the properties of the Bew(,) formula,

$$\mathrm{Bew}(e, d, d) \tag{2}$$

is also a theorem. The only way for both (1) and (2) to be theorems is for the underlying system to be inconsistent.

So far, all we have shown is that our "sufficiently rich" system is either inconsistent or has a strange formula $\mathcal{D}(d)$ which cannot be derivable. That may not be a real problem. Perhaps $\neg\mathcal{D}(d)$ is a theorem of the system. But if that were the case, the formula

$$\exists z\ (\mathrm{Bew}(z, d, d)) \tag{3}$$

is a theorem which asserts the provability of $\mathcal{D}(d)$. But that *still* may not be a real problem, because the formula only asserts that there is an entity that purports to be an encoding of a derivation of $\mathcal{D}(d)$. In an arbitrary interpretation, that entity may not correspond to an actual integer, and there may still be no encoding of a derivation of $\mathcal{D}(d)$.

If we are willing to accept that the standard structure with the natural numbers is a model for our system, then there can be no element z satisfying $\mathrm{Bew}(z, d, d)$,

and formula (3) cannot be a theorem of the system. Therefore, if we accept the natural numbers as a model for our system, then we have shown that *neither* $\mathcal{D}(d)$ nor $\neg\mathcal{D}(d)$ is a theorem.

While it may seem harmless to accept the natural numbers (which, after all, was our intended interpretation) as a model for our system, logicians are leery of making such assumptions. If one is studying the foundations of mathematical reasoning, one ought not assume that all facts about the integers are known. Fortunately, there is a slight twist to the above argument which permits it to be carried out on a strictly syntactical basis, without appeal to models. The result is known as the Gödel-Rosser Incompleteness Theorem. The formula Ψ in that case asserts, very roughly, "If Ψ is provable, then there is a shorter proof of $\neg\Psi$."

Theorem 6.1 (Gödel-Rosser Incompleteness Theorem) *If the logical system $\mathcal{L}$ is sufficiently rich (in the sense described above) and consistent, then there is a formula Δ such that neither Δ nor $\neg\Delta$ is a theorem of $\mathcal{L}$.*

The word "complete" is (unfortunately) used in many different ways in logic. Here, a logical system $\mathcal{L}$ is **complete** if for each closed formula Φ, either Φ or $\neg\Phi$ is a theorem of $\mathcal{L}$. Contrast that definition with the definition of consistency: The system $\mathcal{L}$ is **consistent** if for each formula Φ, *at most* one of Φ and $\neg\Phi$ is a theorem of $\mathcal{L}$. A consistent and complete system has the desirable property that, for each closed formula Φ, *exactly one* of Φ and $\neg\Phi$ is derivable. The Incompleteness Theorem is the disappointing result that only relatively weak systems are both consistent and complete.

One can go extend the proof of the Incompleteness Theorem and produce a formula $\text{Cons}_{\mathcal{L}}$ that asserts that the logical system $\mathcal{L}$ is consistent. By formalizing in the logical system $\mathcal{L}$ all the reasoning we have summarized, one can prove that $\text{Cons}_{\mathcal{L}}$ is not a theorem of $\mathcal{L}$.

Theorem 6.2 (Gödel's Second Incompleteness Theorem) *If the system $\mathcal{L}$ is sufficiently rich and consistent, then neither* $\text{Cons}_{\mathcal{L}}$ *nor* $\neg\text{Cons}_{\mathcal{L}}$ *is a theorem of $\mathcal{L}$.*

You might ask, after coming this far, "What does all this have to do with computer programming?" The answer is buried in the details of the proof, where the notion of "computable function" is formalized. The negative solution to the Halting Problem and the Incompleteness Theorems for logic are both illustrations of one general, underlying principle.

It turns out that any computable function of natural numbers is expressible in the sufficiently rich system $\mathcal{L}$. If $f : \mathbb{N} \to \mathbb{N}$ is a computable function (intuitively, one

that can be calculated with a computer program), then there is a relatively simple formula $R_f(\ ,\ )$ of two free variables such that $f(m) = n$ if and only if $R_f(m, n)$ is a theorem of $\mathcal{L}$. The methods we have described show that the function bew defined by

$$\text{bew}(n) = \begin{cases} 0 & \text{if } n \text{ encodes a theorem of } \mathcal{L}\text{, and} \\ 1 & \text{otherwise} \end{cases}$$

is not similarly representable and therefore not computable. No computable function can tell us whether or not a particular formula has a derivation in $\mathcal{L}$. This result gives us some insight into why it is difficult in systems for predicate logic (like Natural Deduction, the Sequent Calculus, and Resolution) to find derivations of formulas.

7 Temporal Logic

Mathematical truth and facts about computations differ in a fundamental way. Mathematical truth is “timeless,” while computations evolve over time. It is therefore reasonable to attempt to incorporate a notion of time into logics for reasoning about computation.

Logicians have been studying such formal systems, called **temporal logics**, for over a century. There are many variations. We will restrict ourselves to one very simple propositional system.

7.1 Syntax

As with propositional logic, we begin with an infinite set P of propositional letters and the usual connectives $\neg$, $\wedge$, $\vee$, $\Rightarrow$, and $\equiv$. We add two unary connectives $\Box$ and $\Diamond$ and give a natural definition of formula.

1. A propositional letter p is a formula.
2. If A is a formula, then so are $(\neg A)$, $(\Box A)$, and $(\Diamond A)$.
3. If A and B are formulas, then so are $(A \wedge B)$, $(A \vee B)$, $(A \Rightarrow B)$, and $(A \equiv B)$.
4. Nothing else is a formula.

We adopt a standard convention for eliminating unnecessary parentheses: The connectives $\Box$ and $\Diamond$ bind most tightly, followed in order by $\neg$, $\wedge$, $\vee$, $\Rightarrow$, and $\equiv$.

7.2 Semantics

Time is represented as a set $\mathcal{T}$ of moments. There is a rich variety of choices for $\mathcal{T}$. Normally, but not always, $\mathcal{T}$ is linearly ordered. Sometimes there is an initial moment. Here, we adopt the simplest notion of time, where the moments are represented by integers. A larger integer is a "later" time.

Informally, the formula $\Box A$ is interpreted to mean "A is always true," and $\Diamond A$ is interpreted as "A is sometimes (or eventually) true."

In the propositional logic, an interpretation is a function from proposition letters to truth values. Here, in our temporal logic, an **interpretation** is a function v whose domain is $\mathcal{T}$, and for each i in $\mathcal{T}$, $v(i)$ is a function from proposition letters to truth values. Such a function v has the signature

$$\mathcal{T} \to \big(P \to \{\text{False, True}\}\big).$$

Thus, $v(i)(p)$ gives the truth value of the proposition p at the moment i.

As we did with the propositional logic, we extend an interpretation v to a function v' on all formulas so that for each i in $\mathcal{T}$, the function $v'(i)$ maps formulas to truth values. For the logical connectives, we just adopt the usual truth-table definitions; for example,

- $v'(i)(A \wedge B) = \text{True}$ if $v'(i)(A) = v'(i)(B) = \text{True}$, and $v'(i)(A \wedge B) = \text{False}$ otherwise.

For the temporal connectives, we write

- $v'(i)(\Box A) = \text{True}$ if for each j with $i \le j$, $v'(j)(A) = \text{True}$, and $v'(i)(\Box A) = \text{False}$ otherwise.
- $v'(i)(\Diamond A) = \text{True}$ if for some j with $i \le j$, $v'(j)(A) = \text{True}$, and $v'(i)(\Diamond A) = \text{False}$ otherwise.

A formula is **valid** if it is true at all moments in all interpretations. The temporal connectives act like quantifiers over moments in time. It is therefore not surprising that the formula

$$\Box A \equiv \neg\Diamond\neg A$$

is valid. The formula asserts that "A is always true" is equivalent to the negation of "A is sometimes false." In the same spirit, $\Box A \Rightarrow A$, $\Box A \Rightarrow \Box\Box A$, and $\Box(A \Rightarrow B) \Rightarrow (\Box A \Rightarrow \Box B)$ are also valid.

Example 7.1 The formula $\Diamond(p \Rightarrow q) \Rightarrow (\Diamond p \Rightarrow \Diamond q)$ is not valid. Take v to be a valuation in which

$$v(i)(p) = \begin{cases} \text{True} & \text{if } i = 47, \text{ and} \\ \text{False} & \text{otherwise,} \end{cases}$$

and $v(i)(q) = \text{False}$ for all i. Then $v'(0)\big(\Diamond(p \Rightarrow q)\big)$ is True because $v'(i)(p \Rightarrow q)$ is True at some moment i. In fact, it is True at all moments except for $i = 47$.

Also, $v'(0)(\Diamond p)$ is True because $v'(47)(p)$ is True, and $v'(0)(\Diamond q)$ is False. Putting all this together shows that

$$v'(0)\big(\Diamond(p \Rightarrow q) \Rightarrow (\Diamond p \Rightarrow \Diamond q)\big) = \text{False}.$$

Example 7.2 The formula $\Diamond\Box A$ says "eventually A is always true." That is, from some time moment onwards, A is true. Put in another way, the formula says that A can be false only finitely many times.

In contrast, the formula $\Box\Diamond A$ says that it is always the case that A is eventually true, or equivalently, "A is true infinitely often."

Clearly, if A is false only finitely often, then it is true infinitely often. This informal reasoning suggests that $\Diamond\Box A \Rightarrow \Box\Diamond A$ is valid. The validity of the formula can, of course, be established rigorously.

7.3 Deductions

There are seven rules of inference for our temporal logic.

1. $A_1, \cdots, A_k \vdash B$if B is derivable from $A_1, \ldots, A_k$ in propositional logic. We do not specify which propositional system; any one will do. Notice that the formulas may contain temporal connectives. *Modus ponens* is a special case of this rule.

2. $A \vdash \Box A$.

3. $\vdash \Box A \Rightarrow A$.

4. $\vdash \Box A \Rightarrow \Box\Box A$.

5. $\vdash \Box\big(A \Rightarrow B\big) \Rightarrow \big(\Box A \Rightarrow \Box B\big)$.

6. $\vdash \neg\Diamond A \equiv \Box\neg A$.

7. $\vdash \Diamond\Box A \Rightarrow \Box\Diamond A$.

It is easy to show, by induction on the length of a derivation, that any derivable formula is true at all times in all interpretations. Therefore, the system we have presented is sound, relative to our semantics.

Notice that the converse of Rule 3, $A \Rightarrow \Box A$, is not a theorem; there are many counterexamples to its validity. Rule 2 states that, if A is derivable, then so is $\Box A$. This is much weaker than deriving $A \Rightarrow \Box A$. Why?

It is convenient to list one derived rule, which can easily be justified using Rules 2 and 5.

8. $A \Rightarrow B \vdash \Box A \Rightarrow \Box B$.

Example 7.3 By Rule 1, $A \Rightarrow \neg\neg A$ is a theorem. Then by Rule 8, $\Box A \Rightarrow \Box\neg\neg A$ is also a theorem. Similarly, $\Box\neg\neg A \Rightarrow \Box A$ is a theorem. Combining these two results in an additional use of Rule 1 gives the theorem $\Box\neg\neg A \equiv \Box A$.

Example 7.4 We can write derivations informally to capture the underlying reasoning. For example, we prove a counterpart to Rule 6. By Rule 6 itself, the formula

$$\neg\Diamond\neg A \equiv \Box\neg\neg A$$

is a theorem. Combining that with the result of Example 7.3, we obtain

$$\neg\Diamond\neg A \equiv \Box A.$$

Negating both sides and eliminating double negation gives $\Diamond\neg A \equiv \neg\Box A$.

Example 7.5 Given our semantics, we would expect that $\Box A \Rightarrow \Diamond A$ is a theorem. We can derive it as follows:

1. $\Box A \Rightarrow A$, from Rule 3.

2. $A \Rightarrow \neg\neg A$, from the predicate calculus.

3. $\neg\neg A \Rightarrow \neg\Box\neg A$, as the contrapositive of an instance of Rule 3.

4. $\neg\Box\neg A \Rightarrow \Diamond A$, from one direction of Rule 6.

Invoking the transitivity of $\Rightarrow$ to link these together gives $\Box A \Rightarrow \Diamond A$.

Example 7.6 We know that $A \wedge B \Rightarrow A$ is a theorem of the predicate calculus. Applying Rule 8, gives $\Box(A \wedge B) \Rightarrow \Box A$. Similarly, we can derive $\Box(A \wedge B) \Rightarrow \Box B$. Combining these gives

$$\Box(A \wedge B) \Rightarrow \Box A \wedge \Box B. \tag{7}$$

Next, an application of Rule 8 to $A \Rightarrow (B \Rightarrow A \wedge B)$ gives

$$\Box A \Rightarrow \Box(B \Rightarrow A \wedge B).$$

An application of Rule 5 yields

$$\Box(B \Rightarrow A \wedge B) \Rightarrow \big(\Box B \Rightarrow \Box(A \wedge B)\big).$$

By the transitivity of $\Rightarrow$, we obtain $\Box A \Rightarrow \big(\Box B \Rightarrow \Box(A \wedge B)\big)$, or equivalently,

$$\Box A \wedge \Box B \Rightarrow \Box(A \wedge B). \tag{8}$$

Combining Formulas (7) and (8), we obtain

$$\Box A \wedge \Box B \equiv \Box(A \wedge B).$$

The analogous formulas $\Box A \vee \Box B \equiv \Box(A \vee B)$ and $\Diamond A \wedge \Diamond B \equiv \Diamond(A \wedge B)$ are not theorems; see Assignment 8.

Example 7.7 Here is another derivation of $\Box A \Rightarrow \Diamond A$ that uses the result of Example 7.6. We have

$$\Box A \wedge \Box\neg A \Rightarrow \Box(A \wedge \neg A).$$

From Rule 3, we have

$$\Box(A \wedge \neg A) \Rightarrow A \wedge \neg A.$$

The transitivity of $\Rightarrow$ gives us $\Box A \wedge \Box\neg A \Rightarrow A \wedge \neg A$, and its contrapositive

$$\neg(A \wedge \neg A) \Rightarrow \neg(\Box A \wedge \Box\neg A).$$

The hypothesis is a theorem of predicate logic, so we may use *modus ponens* to obtain $\neg(\Box A \wedge \Box\neg A)$, or equivalently, $\Box A \Rightarrow \Diamond A$.

Example 7.8 So far, we have not used Rule 7. In Examples 7.5 and 7.7, we saw that $\Box A \Rightarrow \Diamond A$ is a theorem. Applying Rule 8, gives $\Box\Box A \Rightarrow \Box\Diamond A$. Combining this with Rule 4 produces

$$\Box A \Rightarrow \Box\Diamond A.$$

Although the result looks suspiciously like the formula in Rule 7, there is no way to derive Rule 7 itself from the other rules. Without Rule 7, there would be no way to derive the valid formula $\Diamond\Box A \Rightarrow \Box\Diamond A$.

Example 7.9 The formula $\Box A \wedge \Diamond B \Rightarrow \Diamond(A \wedge B)$ is a theorem. To derive it, apply Rule 5 to obtain

$$\Box(A \Rightarrow \neg B) \Rightarrow \big(\Box A \Rightarrow \Box\neg B\big).$$

Use exchange of hypothesis and then replace the right member with its contrapositive, to produce the theorem

$$\Box A \Rightarrow \big(\neg\Box\neg B \Rightarrow \neg\Box(A \Rightarrow \neg B)\big).$$

Using Rule 6 and its consequence from Example 7.4 gives the formula

$$\Box A \Rightarrow \big(\Diamond B \Rightarrow \Diamond(A \wedge B)\big)$$

which is equivalent to the desired result, $\Box A \wedge \Diamond B \Rightarrow \Diamond(A \wedge B)$.

7.4 Applications

Temporal logic is used to reason about computer programs and hardware. Many problems in networking, processor design, and operating systems involve concurrent actions and are amenable to temporal logic.

One class of desirable properties for software or systems are **safety properties,** which assert "nothing bad happens." A typical example is mutual exclusion. When two processes share memory, we want to insure that, at any given time, one process at a time is in the critical section of code that involves the shared memory. Let p_1 stand for the assertion "process 1 is in its critical section," and let p_2 stand for "process 2 is in its critical section." Then mutual exclusion can be expressed by the temporal formula

$$\Box\neg\big(p_1 \wedge p_2\big).$$

Another class of desirable properties are **liveness properties,** which assert "something good happens." Freedom from deadlock is a liveness property. If the proposition letter r represents the granting of a particular request, then $\Diamond r$ asserts that the request is eventually granted. More realistically, if two programs are sharing the same processor, we can let q_1 be true when the first program is executing and q_2 be true when the second is executing. The temporal formula

$$\Box\Diamond q_1 \wedge \Box\Diamond q_2$$

asserts that each process gets infinitely many time slices on the processor.

As we saw earlier in the course, propositional systems are very weak, and assertions in temporal propositional logic are not very useful. One natural extension is to incorporate variables and quantifiers into the temporal system. One formula might be

$$\forall a \, \Box\big(\text{Request}(a) \Rightarrow \Diamond \, \text{Satisfy}(a)\big),$$

which says that, should a service a be requested, it is eventually satisfied. The $\Box$ operator appears to be sure that the satisfaction does not occur *before* the request.

Typical theorems of a temporal predicate logic would then be

$$\forall x \, \Box\Phi \equiv \Box \, \forall x \, \Phi$$

and

$$\exists x \, \Box\Phi \Rightarrow \Box \, \exists x \, \Phi.$$

In applications like these, it is sometimes useful to strengthen temporal logic with additional connectives. One such connective is a unary connective $\bigcirc$, standing for "next." Then $\bigcirc A$ is true at moment i if A is true at moment $i+1$. With the natural numbers representing moments in time, we have an induction axiom,

$$A \wedge \Box\big(A \Rightarrow \bigcirc A\big) \Rightarrow \Box A.$$

Further, it is sometimes useful to refine the notion of time. One idea is to throw away the starting point and have an infinite past as well as an infinite future. Our system could be adapted by taking all the integers, positive and negative, to represent moments in time. Time that is not linearly ordered is also used.

7.5 Further Reading

The subject of temporal logic is a specialization of **modal logic,** in which the connectives $\Box$ and $\Diamond$ are interpreted as "is necessary that" and "is possible that," respectively. Philosophers are intensely interested in modal systems. For an introduction, see Hughes and Cresswell, *A New Introduction to Modal Logic,* Routledge, 1996. Our system is equivalent to the one that those authors call S4.2.

As suggested by the previous section, temporal predicate logic is used to reason about programs and processors. Many computer scientists have been working in this field. For examples, see Lamport, "The Temporal Logic of Actions," *ACM Transactions on Programming Languages and Systems,* 17(3):507–534, May 1995, and Manna and Pnueli, *Temporal Verification of Reactive Systems: Safety,* Springer-Verlag, 1995.

Appendix A Greek Alphabet

alpha	α	A
beta	β	B
gamma	γ	Γ
delta	δ	Δ
epsilon	ϵ	E
zeta	ζ	Z
eta	η	H
theta	θ	Θ

iota	ι	I
kappa	κ	K
lambda	λ	Λ
mu	μ	M
nu	ν	N
xi	ξ	Ξ
omicron	o	O
pi	π	Π

rho	ρ	P
sigma	σ	Σ
tau	τ	T
upsilon	υ	Υ
phi	ϕ	Φ
chi	χ	X
psi	ψ	Ψ
omega	ω	Ω