A Brief Introduction to Logic

Carlo Tomasi

1 Introduction

Logic attempts to formalize reasoning, and this is useful on many grounds:

Clean Thinking: Formalizing an argument helps discovering possible flaws in it, or convincing us and others of its validity.

Automatic Reasoning: A formal system can be implemented as software on a computer, so there is hope of building machines that reason.

Human Cognition: If we believe that our brains are machines, understanding logic may lead to understanding human thought.

The "human cognition" motive for studying logic is the oldest and loftiest one, and has excited philosophers and logicians for thousands of years. Section 8 returns to this point and briefly summarizes the state of affairs of this fascinating enquiry.

Automatic reasoning has become a thriving area of artificial intelligence and formal verification theory, and has produced computer programs that diagnose diseases, recommend ways to fix broken computers, play chess, schedule the operations of the Hubble telescope, verify airline flight schedules, prove the correctness of software and hardware systems, and much more.

We will study the basics of logic for the "Clear Thinking" reason, because we will be able to use logic in this fashion within this course. However, Section 7 also sketches *inference by resolution* as a preview of automatic reasoning. Hopefully, what you learn about logic here will inspire you to find out about the other aspects of this discipline as well.

The next Section distinguishes between *propositional logic* and *predicate logic*, a traditional layering of the subject into an "easy but somewhat inexpressive" language (propositional logic) and a "harder but powerful" language (predicate logic). Section 3 makes general remarks on *logical inference*, the main reason for having logic in the first place. Section 4 introduces a convenient formalism for describing languages, the *Backus-Naur form*. The basics of propositional and predicate logic are quickly summarized in Sections 5 and 6, and Section 7 sketches resolution. Section 8 overviews what logic can and cannot do.

2 Propositions and Predicates

A *proposition* is a sentence to which it makes sense to attach a value of *true* or *false*. For instance, "Martians are green" is a proposition: we may not know if this sentence is true or false, but the question of whether it is true or false makes sense. An example of a sentence that is not a proposition is "Stop giggling!" Asking if this is true or false is meaningless.

Logic attempts to capture a body of knowledge about some domain in the form of an initial set of statements[1] that are known or assumed to be true. This set is often called the *knowledge base*. Sometimes, the knowledge base is redundant. For instance, it may include a proposition p that states that three is a prime number, another proposition q that states that odd numbers are not divisible by two, and a third proposition r according to which a natural number a cannot divide a natural number b if $a > b > 0$. Then, if the knowledge base contains also basic facts of arithmetic, p can be inferred from q and r: 3 is odd, so it cannot be divisible by 2 (because of q), nor by any other integer greater than 3 (because of r). Because of this, 3 is only divisible (at most) by 1 and itself, and is therefore prime. If we remove r from the base, we do not lose any significant information, since that proposition could be somehow reconstructed from the others.

This example hints at the fact that building a minimal (that is, non-redundant) database can be tricky, since redundancies can be hidden. In fact, it often hard even to ensure *consistency* of the knowledge base: how can we make really sure that no statement in it is incompatible with any other? In one part of the base we may assert that all birds fly, and then inadvertently list penguins as birds somewhere else.[2]

When a knowledge base is pared down to a minimal set of consistent statements, these propositions are called *axioms*. One of the most beautiful and ancient sets of axioms are those Euclid wrote for geometry in his *Elements* in the third century B.C.

In *propositional logic*, propositions are considered *atomic* in the sense that they cannot be broken down into smaller pieces. If the symbol p represents the proposition "Martians are green," then all that logic is allowed to do with p is to assign a truth value to it, either true or false, and to use p to build more complex statements by appending other symbols to p (as in $\neg p$ for "it is not the case that p") or combining p with other propositions through a handful of connectives such as "and," "or," or "if . . . then . . .." These longer statements are called *formulas* or *compound propositions*.[3]

Since formulas in propositional logic are built by combining a finite number of propositions, everything is finite. In principle, one can check truth values of formulas by mechanically building tables that tell how truth values of formulas can be computed from those of their constituent propositions. However, the tables one needs to consider can grow in size very quickly, making *efficient* reasoning far from trivial.

The atomic nature of propositions is a severe limitation of what we can say in propositional logic. For instance, if p still means that "Martians are green" and we also want to say that "Agyr is Martian," all we can do is to introduce a new propositional symbol, say, q to represent this latter observation. The two symbols p and q are somehow connected in our minds, because they both

---

[1]"Statement" here is a more generic term than "proposition." We will be more precise later.

[2]*Non-monotonic logic* studies systems of statements some of which may have to be retracted at some point.

[3]Rosen uses "compound proposition." This note uses the shorter term "formula."

refer to Martians, but there is nothing in the symbols p and q that reflects this connection. In particular, we have no way to infer that Agyr is green, because this would require to somehow break down p into a part that has to do with being Martian and a different part that has to do with being green, with the two parts being connected by the fact that individuals that have the first property also have the second.

In brief, propositional logic is missing a way to express relationships between objects (Agyr) and properties of objects (being green, or Martian). Without this basic element we cannot hope to go very far.

Predicate logic addresses this limitation by introducing objects called *terms*, and statements called *predicates*, which are parameterized by a number of arguments called *variables*. For instance, $P(x)$ is a predicate in the variable x, a placeholder that can be replaced with any term. For instance, $P(x)$ could mean "x is a Martian," and $Q(y, z)$ could mean "y is z's child." A predicate does not have a truth value of its own, but it takes on a truth value when all variables are either replaced by terms (for instance, $P(\text{Agyr})$ means "Agyr is a Martian") or *quantified*. Quantification will be discussed in Section 6. As a preview, $\forall x\, P(x)$, read "For all x it is the case that $P(x)$" is a predicate in which the variable x is said to be *universally quantified*. This quantified predicate in our example would mean that everybody (and everything) is a Martian, a statement that is presumably false, unless Martians are all we ever talk about. The expression $\exists x\, P(x)$, read "There exists an x for which it is the case that $P(x)$" is an example of *existential quantification*. The expression means that "There exists at least one Martian," a statement whose truth value we may not know, but is nonetheless a well defined notion. Just as in propositional logic, *formulas* can be built out of predicates through logical connectives.

Predicates and quantification open vast possibilities of expression, and these come with a price. The main issue is that the *universe* of objects we wish to consider is often infinite. Mars may have a small population, but if we now want to talk about Earthlings our universe counts at least several billion elements. If we want to talk about numbers, we are in even greater trouble. Reasoning becomes much harder, even in principle, in predicate logic, but is still possible as we will see.

So in a nutshell, propositional logic is about atomic facts, while predicate logic is about objects, their properties, and their relationships. Incidentally, a predicate with no arguments is a proposition, so propositional logic is a subset of predicate logic.

3 Inference and Semantics

In both propositional and predicate logic, *inference* computes the truth value of new formulas from the truth values of the formulas in the knowledge base. This computation is sometimes also called *deduction* and applies simple operations out of a short, predefined list of so-called *inference rules*. The chain of inference rules used to deduce a new formula, that is, to show that the formula is true, is called a *proof*, and each of the formulas thus proven is called a *theorem*.

For instance, an inference rule called *Modus Ponens* states that if proposition p is true, and if every time p is true also q is true, then q is true as well: If it is raining, and whenever it rains there are clouds in the sky, then there are clouds in the sky.

Modus Ponens, for one, is a very simple rule, and yet most inference rules are of comparable

complexity. We may think that rules this simple are not going to take us very far. Yet simplicity of the inference rules is the whole point of logic: if the rules are few and simple, we may be able to convince ourselves that they are correct. A proof may be made of a long chain of rule applications, but each step is crystal clear, and our faith in the conclusion is strong.

A sometimes baffling but crucial aspect of logic is that inference is a purely syntactic game, that is, it ignores the meaning of the propositions entirely. Suppose that we know the truth of the following propositions:

- Anyone who attends all lectures and does all the homework gets a passing grade.

- Abe got a failing grade.

- Abe attended all lectures.

After a moment's thought, we conclude that Abe must have skipped some homework. How did we make this inference? Did we use our knowledge about courses for under-achievers and our acquaintance with Abe's sleeping habits?

Let us look at another example:

- Any isosceles triangle with a sixty-degree angle is equilateral.

- Triangle T is not equilateral.

- Triangle T is isosceles.

We can conclude from this that triangle T has no sixty-degree angle. Is our conclusion based on a good understanding of geometry?

Logicians contend that both conclusions above are inescapable, and are based not on an understanding of what the propositions mean (that is, their *semantics*), but purely on their syntactic structure and on a fixed set of inference rules. If we look carefully at the two examples above, we see that in fact their sentences have the same syntactic structure, once we strip away the idiosyncracies of the English language:

- For all x, if $P(x)$ and $Q(x)$ are true then $R(x)$ is true.

- $R(a)$ is false.

- $P(a)$ is true.

And the conclusion is that $Q(a)$ is false. Technically, we say that the three facts above *imply $Q(a)$*. The symbol '$\Rightarrow$' is often used to denote implication.

Check that this structure fits both examples with the following replacements:

- First example:

    - $P(x)$: x attends all lectures
    - $Q(x)$: x does all the homework

– $R(x)$: x gets a passing grade

  – a: Abe

- Second example:

  – $P(x)$: x is an isosceles triangle

  – $Q(x)$: x has a sixty-degree angle

  – $R(x)$: x is equilateral

  – a: triangle T

We will see in Section 7 that the inference technique called *resolution* can in fact prove either argument based only on the syntactic structure of the formulas it involves, and using repeatedly a *single* inference rule. Having done this, we are confident that every argument that has the form above is correct, even if we do not understand the semantics. Suppose for instance that we are told the following facts:

- Anything that stols and worbles galts.

- Jib does not galt.

- Jib stols.

Then we can conclude that Jib does not worble.

Thus, logic handles arguments in a formal, syntactic way. The surprising and exciting part is that this game can in fact prove entire bodies of knowledge. For instance, the British mathematicians and philosophers Alfred North Whitehead and Bertrand Russell proved large chunks of mathematics in an axiomatic way in their epoch-making book *Principia Mathematica*.

As a more widely known example, all of Euclidean geometry can be deduced from a handful of axioms. More recently, entire new geometries have been created by replacing the single axiom of parallels from Euclid's system with modified versions. Euclid's axiom (his Fifth Postulate) states that given a line L and a point P not on L, there is exactly one line through P and parallel to L. In the Nineteenth century, the Russian mathematician Nikolai Ivanovich Lobachevsky developed *hyperbolic geometry* based on the axiom that there are at least two such lines through P, and the German mathematician Georg Friedrich Bernhard Riemann developed *elliptic geometry* based on the axiom that there are no such lines. The question is not, which version of the Fifth Postulate (and thus which geometry) is the true one. As a logical matter, each version of the Fifth Postulate is consistent with the other axioms of geometry, so each geometry is OK.

If you want to make concrete sense of a non-Euclidean geometry you are asking a question of semantics, not of logic, which is not concerned with meaning. For instance, the sphere is a *semantic model* of (two-dimensional) elliptic geometry, that is, it represents a set of objects for which the relationships described by this geometry hold. In the spherical model, lines are great circles, and "parallel" means non-intersecting, just as it does for lines. Pick a great circle on the sphere (say, the Greenwich longitude circle) and a point not on it (say, New York). Two great circles on the sphere always intersect, so any great circle we draw through New York will intersect

the Greenwich circle: there are no great circles through New York that are parallel to the Greenwich circle, in accord with Riemann's version of the Fifth Postulate.

Incidentally, the sphere is a special ellipsoid, and this is why Riemann's geometry is called elliptic. A semantic model of three-dimensional elliptic geometry is the three-dimensional sphere in four dimensions (just as the "standard" sphere is a two-dimensional object that lives in three dimensions), where great circles are replaced by great spheres, three-dimensional spheres of maximal diameter. Do not try to visualize...

A semantic model of two-dimensional hyperbolic geometry, as you might guess, is a hyperboloid. More specifically, one sheet of a two-sheet hyperboloid of revolution serves as the model, on which lines are represented by intersections of planes through the origin with the sheet. This model is known as the Minkowski or Lorentz model, and is a bit tricky to visualize. A simpler model to visualize is the Klein model for two-dimensional hyperbolic geometry, which is the interior of a circle (a disk), with circle chords playing the role of lines. If you pick a point P within the disk, and a chord L that does not go through the disk, there are infinitely many chords through P that do not intersect L (they would if you extended them beyond the disk, but this is outside our world). Both models generalize to more than two dimensions.

4   Backus-Naur Form

As we saw, logic is the study of formal reasoning, but *a* logic is a formal reasoning system, which includes a *language*, that is, a set of symbols and compositional rules that allow expression, and an *inference system* (or *deductive system*) that allows proving sentences in the language.

A convenient and compact way to specify a language is the so-called *Backus-Naur* Form (BNF for short), invented by the American mathematician John Backus and further refined by the Danish computer scientist Peter Naur. Most computer languages today are defined in BNF, starting with Algol, the language Naur wrote a compiler for.

BNF is a *grammar*, that is, a set of *symbols* and *productions*. Symbols, in turn, come in two types: *terminal* and *non-terminal* symbols. The terminal symbols are the signs that actually appear in the language the grammar describes. Non-terminal symbols describe combinations of parts.

BNF productions have the following form:

$$\phi \ ::= \ \phi_1 \ | \ \dots \ | \ \phi_n$$

where ϕ is a non-terminal symbol and ϕ_i is an *alternative*, that is, a sequence of (possibly zero) terminal and (possibly zero) non-terminal symbols. The production above means "any occurrence of ϕ can be replaced by either ϕ_1 or $\dots$ or ϕ_n."

Suppose, for instance, that we want to define a grammar for integer arithmetic. Terminal symbols are digits 0, ..., 9, parentheses (and), and the four operation symbols +, −, x, /. Nonterminal symbols denote the structures one is allowed to build with the terminal symbols. The simplest structure is a positive integer, which is a sequence of one or more digits, the first of which cannot be 0. This definition can be made more compact and clear by the use of BNF.

Here are the BNF productions that make up the grammar for positive integers (a sub-grammar of the grammar for arithmetic):

<positive-integer> ::= *<positive-digit>* | *<positive-integer>* *<digit>*
<digit> ::= 0 | *<positive-digit>*
<positive-digit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Let us understand this, before we continue with arithmetic. The form *<keyword>* is used to denote non-terminal symbols. The keywords used between the '<' and '>' are of purely mnemonic value, since nonterminal symbols are not part of the language itself, just of the grammar. We could have used *<t1>*, *<t2>*, and so forth, but the grammar would be less legible. Terminal symbols like 0 or 4, on the other hand, are represented by themselves. It is useful to use different fonts for terminal and non-terminal symbols, but this is not strictly necessary.

The first time around, the grammar for positive integers given above is probably best understood when read from the bottom up. The production

<positive-digit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

answers the question, How do I make a positive digit? It is either 1 or . . . or 9. This is the simplest type of production to understand, because all symbols in its Right-Hand Side (RHS) are terminal.

One level up, we encounter this production:

<digit> ::= 0 | *<positive-digit>*

Here we find a non-terminal symbol, *<positive-digit>*, on the RHS. However, we know what a positive digit is, so this production merely says that a digit is any digit from 0 to 9. Why not just define a digit directly as follows?

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

This is because zero is a special digit, as it cannot be the first digit of a number, and this distinction is necessary when defining positive integers.

Matters become more interesting if we look at the production for positive integers:

<positive-integer> ::= *<positive-digit>* | *<positive-integer>* *<digit>*

At first, this production seems circular, since the Left-Hand Side (LHS) *<positive-integer>* also appears in the RHS. How do I know what a positive integer is (so I can use one on the RHS), if this is what I am trying to define in the first place (since it is on the LHS)? However, the RHS also offers an alternative replacement for the LHS, namely, a *<positive-digit>*, which I do know how to define. So the first time I build a *<positive-integer>*, I have to use that alternative. For instance, 7 is a *<positive-digit>*, which through this production also becomes a *<positive-integer>*. Once I have a single-digit, positive integer, I can make bigger ones by appending any digit to it, including zero: this is the *<positive-integer>* *<digit>* alternative. This can continue, creating ever longer positive integers. Since the first one is positive, the number zero can never arise from this production.

A production that is defined in terms of itself, like this one, is said to be *recursive*. The alternative that gets the recursion started (*<positive-digit>* in this case) is called the *base case* of the recursion. Every recursion must have a base case, or else it cannot really start.

We could introduce a production that builds integers:

<integer> ::= 0 | *<positive-integer>* | − *<positive-integer>*

An integer is either zero by itself or a positive integer, or a positive integer with a minus sign in front of it. This production is non-recursive. However, this production is unnecessary, because we will consider a minus sign in front of an integer a compound arithmetic formula, rather than just a number, as shown below.

Reading productions from the bottom up, as we have done so far, is more concrete, because one understands simple elements first, and builds more complex ones from them. This is like building a house: first make the two-by-fours and nails and the drywall panels, then use these to build the home.

Considering productions in reverse order is called *recursive thinking*. Let us try this with the rest of the grammar for arithmetic formulas. Instead of asking "how do I use the elements I have to build bigger ones?" let us ask "what is an arithmetic formula?" It is a formula made up by smaller ones, through a change of sign or a binary[4] arithmetic operator +, −, x, or /.

For safety, we define fully parenthesized formulas. That is, instead of writing $3 + 4/2$ we write $(3 + (4/2))$. This both simplifies the grammar and shows explicitly in what order operations are to be performed, without having to rely on precedence rules (such as "a division takes precedence over a sum"). In addition, we need to provide a base case, since the definition we just made up is recursive. That answers the question "What are all integer-arithmetic formulas that are not made of smaller formulas?" The answer: whole numbers. In symbols, the production for a formula is as follows:

<formula> ::= *<whole-number>* | − (*<formula>*)
            | (*<formula>* *<operator>* *<formula>*)
<operator> ::= + | − | x | /

Note that we do not yet know how to make a whole number. This is the essence of recursive thinking: don't sweat the details, just give them a name, and only look at the big picture. You will fill in the details later. And here they are:

<whole-number> ::= 0 | *<positive-integer>*

To summarize, here is the grammar for arithmetic operations:

<formula> ::= *<whole-number>* | − (*<formula>*)
            | (*<formula>* *<operator>* *<formula>*)
<whole-number> ::= 0 | *<positive-integer>*
<positive-integer> ::= *<positive-digit>* | *<positive-integer>* *<digit>*
<digit> ::= 0 | *<positive-digit>*
<positive-digit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<operator> ::= + | − | x | /

---

[4]A *binary* operator combines *two* operands.

For legibility, we pushed the production for operators down the list, since operators are defined in terms of terminal symbols only. Of course, the order in which productions are written is unimportant.

This is all there is to BNF. Simple and powerful.

Here are a couple of exercises: (i) Modify the grammar for integer-arithmetic formulas so that it does not allow division by zero. (ii) Write a grammar for numbers written in scientific notation.

5 Propositional Logic

Table 1 shows a BNF grammar for propositional logic.

<table>
<tr><td><formula></td><td>::=</td><td><proposition> | (∼ <formula>)</td></tr>
<tr><td></td><td></td><td>| (<formula> <binary-operator> <formula>)</td></tr>
<tr><td><proposition></td><td>::=</td><td>$\mathbb{T}$ | $\mathbb{F}$ | p | q | r | p <whole-number></td></tr>
<tr><td><binary-operator></td><td>::=</td><td>∧ | ∨ | →</td></tr>
</table>

<whole-number> ::= 0 | <positive-integer>
<positive-integer> ::= <positive-digit> | <positive-integer> <digit>
<digit> ::= 0 | <positive-digit>
<positive-digit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Operator Priority Rule: Top-level parentheses and parentheses that are made redundant by the following operator priority are optionally omitted: ∼ binds more tightly than ∧ or ∨, and these bind more tightly than →.

Table 1: A BNF grammar for propositional logic, together with an operator precedence rule to reduce the number of parentheses for greater legibility.

The definition of <whole-number> is borrowed from Section 4. Other than that, the entire language of propositional logic can be specified by three lines in BNF.

Propositions, defined in the second line, are merely symbols. The two special symbols $\mathbb{T}$ and $\mathbb{F}$ denote respectively a *tautology*, that is, a proposition that is always true, and a *contradiction*, one that is always false. Thereafter, the three symbols p, q, and r are letters that stand for propositions, and the last option p <whole-number> provides an infinite supply of propositional names: $p0$, $p1$, and so forth.

Binary operators, defined in the third line, allow connecting propositions into formulas, and formulas into more complex formulas, as defined in the last alternative of the definition of <formula> (first line in Table 1). The names of these operators are 'and' for ∧, 'or' for ∨, and 'conditional' for →. The meaning of these operators is supposed to correspond loosely to the meaning of their

English language counterparts. Their precise meaning will be defined later on. The unary[5] operator ~ in the second alternative for *<formula>* is called 'not'.

5.1 Truth Tables

Propositional logic is concerned with how the truth or falsehood of formulas can be computed from those of its constituent propositions, or from those of other formulas. The rules for computing truth values are specified by one *truth table* for each of the unary or binary operators. Recall that "unary" and "binary" refers to the number of formulas (or propositions, which are formulas as well!) an operator combines. For convenience, we also think of the $\mathbb{T}$ and $\mathbb{F}$ propositions as the result of applying an operator with zero arguments to, well, nothing, so we can have very simple truth tables for them as well.

The truth table for an operator specifies the truth value for every possible combination of truth values that the arguments can take on. So an operator with zero arguments has a truth table with one entry. A unary operator has a truth table with two entries, because the argument can be either true or false. A binary operator has a table with four entries, since for each of the two possible truth values for the first argument there are two possible truth values for the second. Truth tables for all the operators of propositional logic are shown in Table 2. Note the distinction between the propositions $\mathbb{T}$ and $\mathbb{F}$ and the truth values T and F. Please verify that the first five truth tables correspond to the intuitive meaning of the corresponding English connectives.

$\mathbb{T}$
T

$\mathbb{F}$
F

ϕ	$\sim\phi$
T	F
F	T

ϕ	ψ	$\phi \wedge \psi$
T	T	T
T	F	F
F	T	F
F	F	F

ϕ	ψ	$\phi \vee \psi$
T	T	T
T	F	T
F	T	T
F	F	F

ϕ	ψ	$\phi \rightarrow \psi$
T	T	T
T	F	F
F	T	T
F	F	T

Table 2: Truth tables for the operators of propositional logic.

The table for conditional is less immediate. In the *conditional*

$$\phi \rightarrow \psi \,,$$

read "if ϕ then ψ," the formula ϕ is called the *antecedent* or *premise*, and the formula ψ is called the *consequent* or *conclusion*. The conditional is taken to mean that the conclusion is never false

---

[5]A *unary* operator takes *one* argument.

when the premise is true. In other words, $\phi \rightarrow \psi$ it always false when ϕ is true and ψ is false. This is what the second line in the truth table says, so this poses no problem.

The other lines of the truth table for conditional could in principle be defined in different ways, depending on how one interprets the sentence "if ϕ then ψ." However, the definition of conditional given in Table 2 (which is the only one used in logic) is the only definition that is consistent with what this connective is designed for. Specifically, let ϕ represent a set of assumptions about the world. For instance, ϕ could be the set of axioms of Euclidean geometry. Let ψ be an additional statement about the world, for instance, "an isosceles triangle with a sixty-degree angle is equilateral" (we encountered this statement in Section 3).

Let us now suppose that in every set of circumstances in which one assumes ϕ to hold, ψ cannot fail to hold, either: if ϕ is true, then ψ must be true as well. In this case, we say that ϕ *implies* ψ, and write $\phi \Rightarrow \psi$. Well, the conditional operator is designed in such a way that implication is equivalent to conditional being a tautology:

$$\phi \Rightarrow \psi \quad \text{is the case if and only if} \quad \phi \rightarrow \psi \quad \text{is always true.}$$

Verify that you understand this: if ψ is true whenever ϕ is, then we obtain the first row of the truth table of conditional, so $\phi \rightarrow \psi$ is true. The second row never occurs. When ϕ is false, the truth table for conditional yields T regardless of the truth value of ψ. In summary, all combinations of ϕ and ψ that actually occur yield T for the conditional: the conditional $\phi \rightarrow \psi$ is a tautology.

In other words, the conditional operator is designed so that one can check for implication between ϕ and ψ by verifying that $\phi \rightarrow \psi$ is always true. This leads to the first two rows of the truth table for conditional: Under the assumption that ϕ represents a consistent set of statements, the truth value of $\phi \rightarrow \psi$ coincides with the truth value of ψ. On the other hand, it can be shown that anything can be made to follow from a contradictory set of statements, so when ϕ is false ψ will follow in all cases: this yields the last two rows of the truth table.

A simple way to remember the truth tables for the three binary operators is as follows:

and ($\wedge$): True iff[6] both arguments are true.

or ($\vee$): False iff both arguments are false.

conditional ($\rightarrow$): False iff the premise is true and the conclusion is false.

5.2 Compound Truth Tables

The truth tables defined in section 5.1 are all that is needed to compute the truth value of formulas given the truth values of its constituents. For instance, suppose that we want to determine the truth value of

$$\sim p \vee q$$

when p is true and q is false. Then, we would use the truth table for "not" and determine that $\sim p$ is false. From this, and from the "false" value for q, the truth table for "or" would tell us that the formula above is false.

---

[6]"Iff" means "if and only if."

However, it is often useful to keep track of the truth value of formulas for *all* possible combinations for the truth values of the constituent propositions. Most importantly, this need arises when trying to determine whether two formulas actually mean the same thing. Specifically, two formulas are said to be *logically equivalent* when their truth values coincide for all possible combinations of truth values of their constituent propositions.

Then, it becomes useful to build *compound truth tables* for entire formulas as the propositions that appear in them take on all possible values. These tables are best built in stages, with one column per operator. For instance, the compound truth table for the formula above is as follows:

p	q	$\sim p$	$\sim p \vee q$
T	T	F	T
T	F	F	F
F	T	T	T
F	F	T	T

By comparing this table with the one for conditional, we see that the last columns coincide: $\phi \to \psi$ is logically equivalent to $\sim\phi \vee \psi$ (of course, what holds for propositions also holds for formulas).

The truth table we just wrote has four rows because two propositions appear in the formula. With three propositions, the table would have eight rows. Generally, *a compound truth table for a formula with n propositions has 2^n rows.* This is an important observation, because the number 2^n grows fast with n. Truth tables are conceptually simple, but they can become very large.

To facilitate comparisons of truth tables, it is useful to write the truth value combinations in a systematic order. Here are the first n columns for truth tables with n propositions, for $n = 1, 2, 3$:

p
T
F

p	q
T	T
T	F
F	T
F	F

p	q	r
T	T	T
T	T	F
T	F	T
T	F	F
F	T	T
F	T	F
F	F	T
F	F	F

Notice the pattern: the table for n propositions is built by making two copies of the table for $n-1$ propositions, appending a column with all T to the left of the first copy and a column with all F to the left of the second copy, and placing the second copy below the first.

We used truth tables above to show that $p \to q$ and $\sim p \vee q$ are logically equivalent or, with commonly-used notation,

$$p \to q \equiv \; \sim p \vee q$$

Comparison of the appropriate columns in the tables below (shown in bold font) prove the following useful logical equivalences. The first two are called De Morgan's laws. The third is the law

of *contrapositives* (the expression on the right is called the contrapositive of that on the left, and viceversa). The last two equivalences show that both $\wedge$ and $\vee$ distribute with respect to each other.

$$\sim (p\vee q) \equiv \sim p\wedge \sim q$$
$$\sim (p\wedge q) \equiv \sim p\vee \sim q$$
$$p \rightarrow q \equiv \sim q \rightarrow \sim p$$
$$p\vee (q\wedge r) \equiv (p\vee q)\wedge (p\vee r)$$
$$p\wedge (q\vee r) \equiv (p\wedge q)\vee (p\wedge r) .$$

Note the symmetry between the two distribution laws (the last two equivalences). In arithmetic, matters are not as nice: multiplication distributes with respect to addition, but not viceversa: $a(b+c) = ab + bc$, but $a + (bc) \neq (a + b)(a + c)$ (check if you are not sure!).

Here are the truth tables that prove the equivalences above (each table is used for two formulas):

p	q	$p\vee q$	$\sim(p\vee q)$	$\sim p$	$\sim q$	$\sim p\wedge\sim q$
T	T	T	**F**	F	F	**F**
T	F	T	**F**	F	T	**F**
F	T	T	**F**	T	F	**F**
F	F	F	**T**	T	T	**T**

p	q	$p\wedge q$	$\sim(p\wedge q)$	$\sim p$	$\sim q$	$\sim p\vee\sim q$
T	T	T	**F**	F	F	**F**
T	F	F	**T**	F	T	**T**
F	T	F	**T**	T	F	**T**
F	F	F	**T**	T	T	**T**

p	q	$p\rightarrow q$	$\sim p$	$\sim q$	$\sim q\rightarrow p$
T	T	**T**	F	F	**T**
T	F	**F**	F	T	**F**
F	T	**T**	T	F	**T**
F	F	**T**	T	T	**T**

p	q	r	$q\wedge r$	$p\vee(q\wedge r)$	$p\vee q$	$p\vee r$	$(p\vee q)\wedge(p\vee r)$
T	T	T	T	**T**	T	T	**T**
T	T	F	F	**T**	T	T	**T**
T	F	T	F	**T**	T	T	**T**
T	F	F	F	**T**	T	T	**T**
F	T	T	T	**T**	T	T	**T**
F	T	F	F	**F**	T	F	**F**
F	F	T	F	**F**	F	T	**F**
F	F	F	F	**F**	F	F	**F**

p	q	r	$q\vee r$	$p\wedge(q\vee r)$	$p\wedge q$	$p\wedge r$	$(p\wedge q)\vee(p\wedge r)$
T	T	T	T	**T**	T	T	**T**
T	T	F	T	**T**	T	F	**T**
T	F	T	T	**T**	F	T	**T**
T	F	F	F	**F**	F	F	**F**
F	T	T	T	**F**	F	F	**F**
F	T	F	T	**F**	F	F	**F**
F	F	T	T	**F**	F	F	**F**
F	F	F	F	**F**	F	F	**F**

5.3 Inference in Propositional Logic

Suppose now that we want to check if a set Φ of formulas implies another formula ψ:

$$\Phi \Rightarrow \psi \ .$$

In other words, is it the case that whenever the formulas in Φ are true the formula ψ must follow?

Consider as an example the following situation. You are walking to a party with your friend Jane, and you start gossiping about your common friends. "Tom detests being with both Carl and Nathan at the same time – you say – because the two of them always talk to each other when they are together, and they tend to ignore Tom. So if both Carl and Nathan come to tonight's party, then Tom won't be there. Peter is another peculiar character: He likes Heather, but he is shy, and he is comfortable with her only if either of his friends Carl or Samantha are there as well, so they can all chit-chat together. So if Heather comes to the party but neither Carl nor Samantha can come, Peter will stay at home, too. I know that Heather will be there, but Samantha is sick at home. I wonder who we are going to find there."

"Oh – says Jane – then I hope Nathan and Tom are not both at the party tonight, because then Peter will stay at home. I so much wanted to dance with him!"

As you open the door into the dancing hall and let Jane in, you look at her in bewilderment. Is she right? Is it the case that if what you said is all true then Jane is correct?

People are still trickling in, so while Jane chats with Isabel, you sit down at a table, pull a pen out of your pocket, and start scribbling on a napkin.

You call Φ the set of facts you explained to Jane, and ψ what she answered back. You know enough from your first few discrete math classes to know that logic ignores judgement values, wishes, motivations, and explanations, so you summarize just the facts, to which you can attach truth values. To make things manageable, you write down the following propositional symbols:

c**:** Carl goes to the party

n**:** Nathan goes to the party

t**:** Tom goes to the party

h**:** Heather goes to the party

s**:** Samantha goes to the party

p**:** Peter goes to the party

You wish you had studied predicate logic, because then you could have introduced a single predicate $P(x)$, meaning "x goes to the party," and you could have replaced x with terms for each of your friends. But you haven't covered that in class yet. Here are the facts in Φ:

$(c \wedge n) \rightarrow \sim t$**:** If both Carl and Nathan go to the party, Tom stays at home.

$h \wedge \sim (s \vee c) \rightarrow \sim p$**:** If Heather is there but neither Samantha nor Carl are there, then Peter stays home.

$h \land \sim s$: Heather will be there, but Samantha will not.

Of course, these facts are all true, so their set can be replaced by a single conjunction ϕ:

$$((c \land n) \to \sim t) \land (h \land \sim (s \lor c) \to \sim p) \land (h \land \sim s) .$$

And here is what Jane said, ψ:

$$(n \land t) \to \sim p$$

that is, if both Nathan and Tom go to the party, Peter stays at home.

To show implication, that is,

$$\phi \Rightarrow \psi ,$$

all we need to show is that for all combinations of truth values for the six propositions c, n, t, h, s, p that make ϕ true also ψ is true. What happens for all other combinations of truth values does not matter.

In principle, the simplest way to do this is to build a compound truth table for both ϕ and ψ, and then verify that in every row in which ϕ is true ψ is true as well. In practice, this is quite a bit of work, the result of which is shown in Tables 3 and 4. To make the tables fit on their pages, the following abbreviations were used (α indicates an "and," ω an "or," and τ an "if . . . then . . ."):

$$
\begin{aligned}
\alpha_1 &\equiv c \land n \\
\alpha_2 &\equiv h \land \sim s \\
\omega_1 &\equiv s \lor c \\
\alpha_3 &\equiv n \land t \\
\alpha_4 &\equiv h \land \sim \omega_1 \equiv h \land \sim (s \lor c) \\
\tau_1 &\equiv \alpha_1 \to \sim t \equiv (c \land n) \to \sim t \\
\tau_2 &\equiv \alpha_4 \to \sim p \equiv h \land \sim (s \lor c) \to \sim p \\
\phi &\equiv \tau_1 \land \alpha_2 \land \tau_2 \equiv ((c \land n) \to \sim t) \land (h \land \sim (s \lor c) \to \sim p) \land (h \land \sim s) \\
\psi &\equiv \alpha_3 \to \sim p \equiv (n \land t) \to \sim p .
\end{aligned}
$$

Bold entries in the last three columns emphasize the main point: whenever ϕ is true, so is ψ or, which is the same, the conditional $\phi \to \psi$ (recall that Φ and ϕ are interchangeable). Therefore, ϕ implies ψ, and Jane was right.

This always works, so for propositional logic we could in principle stop here. However, you see the problem: It isn't just that filling out these huge tables does not quite look like "inference" in the way we humans would actually reason through a situation like this. Even if we accept this style of "thinking," with just six propositional variables the truth table has $2^6 = 64$ rows, and with fifteen columns (the last column in the tables is there just to clarify a point) there are about 1,000 entries to fill.[7] The exponential growth of the table size with the number of variable makes this brute-force proof technique practical only for inference problems of trivial size.

---

[7]In case you wonder, the tables 3 and 4 were produced with a computer program.

c	n	t	h	s	p	α_1	α_2	ω_1	α_3	α_4	τ_1	τ_2	ϕ	ψ	$\phi \to \psi$
T	T	T	T	T	T	T	F	T	T	F	F	T	F	F	**T**
T	T	T	T	T	F	T	F	T	T	F	F	T	F	T	**T**
T	T	T	T	F	T	T	T	T	T	F	F	T	F	F	**T**
T	T	T	T	F	F	T	T	T	T	F	F	T	F	T	**T**
T	T	T	F	T	T	T	F	T	T	F	F	T	F	F	**T**
T	T	T	F	T	F	T	F	T	T	F	F	T	F	T	**T**
T	T	T	F	F	T	T	F	T	T	F	F	T	F	F	**T**
T	T	T	F	F	F	T	F	T	T	F	F	T	F	T	**T**
T	T	F	T	T	T	T	F	T	F	F	T	T	F	T	**T**
T	T	F	T	T	F	T	F	T	F	F	T	T	F	T	**T**
T	T	F	T	F	T	T	T	T	F	F	T	T	**T**	**T**	**T**
T	T	F	T	F	F	T	T	T	F	F	T	T	**T**	**T**	**T**
T	T	F	F	T	T	T	F	T	F	F	T	T	F	T	**T**
T	T	F	F	T	F	T	F	T	F	F	T	T	F	T	**T**
T	T	F	F	F	T	T	F	T	F	F	T	T	F	T	**T**
T	T	F	F	F	F	T	F	T	F	F	T	T	F	T	**T**
T	F	T	T	T	T	F	F	T	F	F	T	T	F	T	**T**
T	F	T	T	T	F	F	F	T	F	F	T	T	F	T	**T**
T	F	T	T	F	T	F	T	T	F	F	T	T	**T**	**T**	**T**
T	F	T	T	F	F	F	T	T	F	F	T	T	**T**	**T**	**T**
T	F	T	F	T	T	F	F	T	F	F	T	T	F	T	**T**
T	F	T	F	T	F	F	F	T	F	F	T	T	F	T	**T**
T	F	T	F	F	T	F	F	T	F	F	T	T	F	T	**T**
T	F	T	F	F	F	F	F	T	F	F	T	T	F	T	**T**
T	F	F	T	T	T	F	F	T	F	F	T	T	F	T	**T**
T	F	F	T	T	F	F	F	T	F	F	T	T	F	T	**T**
T	F	F	T	F	T	F	T	T	F	F	T	T	**T**	**T**	**T**
T	F	F	T	F	F	F	T	T	F	F	T	T	**T**	**T**	**T**
T	F	F	F	T	T	F	F	T	F	F	T	T	F	T	**T**
T	F	F	F	T	F	F	F	T	F	F	T	T	F	T	**T**
T	F	F	F	F	T	F	F	T	F	F	T	T	F	T	**T**
T	F	F	F	F	F	F	F	T	F	F	T	T	F	T	**T**

Table 3: Part I of the truth table needed to check that $\phi \Rightarrow \psi$. Wherever ϕ is true, so is ψ (bold entries in the ϕ and ψ columns). Equivalently, the conditional $\phi \to \psi$ is a tautology (the last column is all T).

c	n	t	h	s	p	α_1	α_2	ω_1	α_3	α_4	τ_1	τ_2	ϕ	ψ	$\phi \to \psi$
F	T	T	T	T	T	F	F	T	T	F	T	T	F	F	**T**
F	T	T	T	T	F	F	F	F	T	T	F	T	T	F	**T**
F	T	T	T	F	T	F	T	F	T	T	T	F	F	F	**T**
F	T	T	T	F	F	F	T	F	T	T	T	T	**T**	**T**	**T**
F	T	T	F	T	T	F	F	T	T	F	T	T	F	F	**T**
F	T	T	F	T	F	F	F	T	T	F	T	T	F	T	**T**
F	T	T	F	F	T	F	F	F	T	F	T	T	F	F	**T**
F	T	T	F	F	F	F	F	F	T	F	T	T	F	T	**T**
F	T	F	T	T	T	F	F	T	F	F	T	T	F	T	**T**
F	T	F	T	T	F	F	F	T	F	F	T	T	F	T	**T**
F	T	F	T	F	T	F	T	F	F	T	T	F	F	T	**T**
F	T	F	T	F	F	F	T	F	F	T	T	T	**T**	**T**	**T**
F	T	F	F	T	T	F	F	T	F	F	T	T	F	T	**T**
F	T	F	F	T	F	F	F	T	F	F	T	T	F	T	**T**
F	T	F	F	F	T	F	F	F	F	F	T	T	F	T	**T**
F	T	F	F	F	F	F	F	F	F	F	T	T	F	T	**T**
F	F	T	T	T	T	F	F	T	F	F	T	T	F	T	**T**
F	F	T	T	T	F	F	F	T	F	F	T	T	F	T	**T**
F	F	T	T	F	T	F	T	F	F	T	T	F	F	T	**T**
F	F	T	T	F	F	F	T	F	F	T	T	T	**T**	**T**	**T**
F	F	T	F	T	T	F	F	T	F	F	T	T	F	T	**T**
F	F	T	F	T	F	F	F	T	F	F	T	T	F	T	**T**
F	F	T	F	F	T	F	F	F	F	F	T	T	F	T	**T**
F	F	T	F	F	F	F	F	F	F	F	T	T	F	T	**T**
F	F	F	T	T	T	F	F	T	F	F	T	T	F	T	**T**
F	F	F	T	T	F	F	F	T	F	F	T	T	F	T	**T**
F	F	F	T	F	T	F	T	F	F	T	T	F	F	T	**T**
F	F	F	T	F	F	F	T	F	F	T	T	T	**T**	**T**	**T**
F	F	F	F	T	T	F	F	T	F	F	T	T	F	T	**T**
F	F	F	F	T	F	F	F	T	F	F	T	T	F	T	**T**
F	F	F	F	F	T	F	F	F	F	F	T	T	F	T	**T**
F	F	F	F	F	F	F	F	F	F	F	T	T	F	T	**T**

Table 4: Part II of the truth table needed to check that $\phi \Rightarrow \psi$. Wherever ϕ is true, so is ψ (bold entries in the ϕ and ψ columns). Equivalently, the conditional $\phi \to \psi$ is a tautology (the last column is all T).

5.4 Soundness and Completeness

Thus, in propositional logic one could in principle prove everything with truth tables. However, the size of truth tables grows exponentially fast with the number of propositional symbols contained in the formula to be proven. Because of this growth, even in propositional logic it is useful to find inference rules that allow proving formulas more efficiently. These rules replace formulas in the knowledge base with other formulas until the formula to be proven is obtained. This sequence of formula transformations is called a proof, and the resulting formula is said to be *provable* with the given set of inference rules.

Given a set of inference rules and a knowledge base, it is usually not obvious that the inference rules are powerful enough to allow proving every formula the knowledge base implies. In other words, provability and implication are different concepts. However, some sets of inference rules can be shown to be *complete*, in the sense that they can prove anything that is entailed by the knowledge base.

Conversely, it would be disastrous if a set of inference rules allowed to prove formulas that the knowledge base does not imply. To avoid this, inference rules are required to be *sound*, that is, any formula that they produce must be entailed by the formulas they assume to exist in the knowledge base.

For instance, Modus Ponens, the inference rule we considered earlier, assumes a formula ϕ to be true, and assumes that every time ϕ is true also some other formula ψ is true.[8] If formulas ϕ and $\phi \rightarrow \psi$ are in the database, then Modus Ponens authorizes us to add formula ψ to the knowledge base. To show that Modus Ponens is sound, we merely need to recall the truth table for conditional:

ϕ	ψ	$\phi \rightarrow \psi$
T	T	T
T	F	F
F	T	T
F	F	T

The formulas ϕ (first column) and $\phi \rightarrow \psi$ (third column) are simultaneously true only in the first row, and ψ is true in this row as well. Therefore, Modus Ponens is sound: whenever both ϕ and $\phi \rightarrow \psi$ are true, so is ψ, which can then be safely added to the knowledge base.

When we move to predicate logic, there are even deeper reasons for developing inference methods beyond truth tables: In predicate logic, the truth tables become often *infinite* in size.

Rather than discussing inference methods here, and then discussing them again for predicate logic, we now turn to a definition of the language of predicate logic, and we develop directly an inference technique, resolution, for this language in Section 7. Since propositional logic is a subset of predicate logic, we will be able to use resolution even for the former.

6 Predicate Logic

The language of predicate logic expresses properties of objects. The objects are represented by expressions called *terms*, and the properties are represented by expressions called *predicates*. These

---

[8]In Section 3, Modus Ponens was stated for propositions, rather than formulas.

can be defined in BNF as shown in Table 5.

<formula>   ::= *<predicate>* | (~ *<formula>*)
             | (*<formula>* *<binary-operator>* *<formula>*)
             | (*<quantifier>* *<variable>* *<formula>*)

<predicate>   ::= *<predicate-name>* | *<predicate-name>* *<argument-list>*

<function>   ::= *<function-name>* *<argument-list>*
<argument-list>   ::= (*<term-list>*)
<term-list>   ::= *<term>* | *<term-list>* , *<term>*
<term>   ::= *<variable>* | *<constant>* | *<function>*

<binary-operator>   ::= $\wedge$ | $\vee$ | $\rightarrow$
<quantifier>   ::= $\exists$ | $\forall$

<variable>   ::= x | y | z | x *<whole-number>*
<constant>   ::= a | b | c | c *<whole-number>*
<function-name>   ::= f | g | h | f *<whole-number>*
<predicate-name>   ::= P | Q | R | P *<whole-number>*

<whole-number>   ::= 0 | *<positive-integer>*
<positive-integer>   ::= *<positive-digit>* | *<positive-integer>* *<digit>*
<digit>   ::= 0 | *<positive-digit>*
<positive-digit>   ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Operator Priority Rule: Top-level parentheses and parentheses that are made redundant by the following operator priority are optionally omitted: $\sim$, $\exists$, and $\forall$ bind more tightly than $\wedge$ or $\vee$, and these bind more tightly than $\rightarrow$.

Table 5: A BNF grammar for predicate logic, together with an operator precedence rule to reduce the number of parentheses for greater legibility.

Here are a few remarks on these expressions. The last ten lines merely introduce symbol names: The *existential quantifier* $\exists$ and the universal quantifier $\forall$ introduced in Section 2, the usual logical operators, and four families of names for variables, constants, functions and predicates. As usual, a few letters are reserved for each type of names for convenience (for instance, a,b,c for constants), and then an infinite supply of symbols ($c_0, c_1, \ldots$) is provided through the customary production for *<whole-number>* s.

Terms are what predicates can talk about. If no variable appears anywhere in a term, then the term denotes a specific object. A term without variables is called a *ground term*, and is either a

constant, or a function of (functions of) constants. For instance, c is a constant (a ground term) that may perhaps denote Tom, and $f(c)$ (also a ground term) might denote Tom's sibling. *Variables* are placeholders for terms. For instance, x denotes "something" (or "someone;" in either case, not a ground term), and $f(x)$ (not a ground term) is someone's sibling.

Note that the definitions of *<term>*, *<function>*, *<argument-list>* and *<term-list>* are *mutually recursive*, that is, their definitions refer to each other in a circular way: *<term>* refers to *<function>* which refers to *<argument-list>* which refers to *<term-list>* which refers back to *<term>*. However, there are ways out of the loop (that is, there are base cases for the recursion), since a term can also be a *<variable>* or a *<constant>*.

Functions and predicates are formally the same, except for the letters that can be used for their names. However, predicates are combined into formulas (see the first line in Table 5), while functions are used to create terms. Think of predicates as statements (with a verb), and of functions as objects related to other objects (the function's arguments). "Tom is Jack's brother" is a predicate of the form $P(\text{Tom}, \text{Jack})$, while "Jack's brother" is a function, $f(\text{Jack})$. Note the formal abuse: the grammar in Table 5 does not allow using 'Tom' or 'Jack' as constants. This abuse is convenient, so we'll keep doing this. Of course, one could extend the definition of *<constant>* to include the entire English vocabulary. Logic would not become stronger or more general for this.

The top line in Table 5 shows that a *<formula>* is either a *<predicate>* (base case), or a set of formulas connected by logical connectives, just as for propositional logic, or a quantified formula. In the expression

$$(\ \textit{<quantifier>} \ \ \textit{<variable>} \ \ \textit{<formula>} \ ) \ ,$$

the *<formula>* is the *scope* of the quantified *<variable>*. The grammar in Table 5 allows for formulas like the following:

$$P(a,y) \quad , \quad (P(x) \rightarrow Q(x, f(c), g(x))) \quad , \quad (\forall x(\exists y P(x,y,z))) \ .$$

In these formulas, some variables are *free*, that is, they are not quantified: y in the first formula, x in the second, and z in the third. Variables that are not free are said to be *bound*. For instance, x and y in the third formula above are bound. A formula where all variables are bound is a (possibly very complex) statement about (possibly infinitely many) specific objects, so it can in principle be assigned a truth value, and becomes a *closed formula*. A formula that is not closed is said to be *open*.

As noted earlier, the simplest formulas are predicates with no argument, which are called *atomic propositions*. By restricting the grammar for predicate logic in Table 5 to atomic propositions (that is, to the first alternative in the production for *<predicate>*, so that terms, variable, and constants become useless), we obtain the grammar of propositional logic in Table 1 (check this). Thus, propositional logic is a subset of predicate logic.

The formulas shown above are fully parenthesized. We will often omit redundant parentheses and write the second and third formula above, for instance, as follows:

$$P(x) \rightarrow Q(x, f(c), g(x)) \quad , \quad \forall x \exists y P(x,y,z) \ .$$

6.1 Quantification

The main price to pay for the power of the existential and universal quantifier is that truth tables can no longer be used with them, because with an infinite object universe the truth tables would have to be infinitely large. However, it useful to refer to a finite universe as a heuristic device to understand quantification.

Specifically, an easy way to understand and reason about quantifiers is to think of the existential quantifier $\exists$ as a possibly infinite disjunction (a sequence of "or"), and of the universal quantifier $\forall$ as a possibly infinite conjunction (a sequence of "and") over the entire universe of objects. If our universe were to consist of just two objects denoted by the constants a and b, then

$$\exists x P(x) \quad \text{would mean} \quad P(a) \vee P(b)$$

and

$$\forall x P(x) \quad \text{would mean} \quad P(a) \wedge P(b) .$$

This is obvious: if there exists some object such that the predicate $P(x)$ holds for it, then either $P(a)$ or $P(b)$ holds, since a and b are the only two objects we can talk about. Similarly, if the predicate $P(x)$ holds for every object, then both $P(a)$ and $P(b)$ hold.

This device allows finding useful logical equivalences for formulas with quantifiers. For instance, the formula

$$\sim \forall x P(x)$$

can be translated (in our mini-universe) to the following:

$$\sim (P(a) \wedge P(b)) .$$

We can now transform this formula by one of De Morgan's laws to obtain

$$\sim P(a) \vee \sim P(b)$$

which translates back to

$$\exists x \sim P(x) .$$

This chain of translations shows the following logical equivalence:

$$\sim \forall x P(x) \equiv \exists x \sim P(x) .$$

Similarly, one can follow this chain of transformations:

$$\sim \exists x P(x)$$
$$\sim (P(a) \vee P(b))$$
$$\sim P(a) \wedge \sim P(b)$$
$$\forall x \sim P(x) .$$

This shows the following equivalence:

$$\sim \exists x P(x) \equiv \forall x \sim P(x) .$$

The two equivalences just derived extend De Morgan's laws to quantification:

$$\sim \forall x P(x) \equiv \exists x \sim P(x) \quad \text{and} \quad \sim \exists x P(x) \equiv \forall x \sim P(x) \, .$$

Formally these laws state that the "not" operator $\sim$ can be switched with an adjacent quantifier by simultaneously changing the type of quantifier. For instance, $\sim\exists x$ can be replaced with $\forall x\sim$. This is intuitive. For example if it is not the case that there exists an object for which predicate $P(x)$ holds, that amounts to saying that for every object the negation of $P(x)$ holds. Similarly, if it is not the case that for every object $P(x)$ holds, then there exists at least one object for which the negation of $P(x)$ holds.

We have derived these formulas for a universe with two objects. Of course, the transformations still apply to larger universes (to convince yourself, re-derive these equivalences for an universe with three objects).

Since "and" and "or" are associative and commutative (this can be proven with truth tables), we can use our heuristic to prove that nested existential quantifiers can be permuted with each other, and so can nested universal quantifiers:

$$\forall x \forall y P(x,y) \equiv \forall y \forall x P(x,y) \quad \text{and} \quad \exists x \exists y P(x,y) \equiv \exists y \exists x P(x,y) \, .$$

However, existential and universal quantifiers cannot be switched with each other: $\forall x \exists y P(x,y)$ is *not* the same as $\exists x \forall y P(x,y)$, just as $(p\lor q)\land(r\lor s)$ is not the same as $(p\land q)\lor(r\land s)$ (again, truth tables can show this). This is not surprising: to say that every son has a father is not the same as saying that there is one person who is every son's father (that would be a large family indeed).

6.2 Implication in Predicate Logic

In predicate logic, it is generally not possible to use truth tables to establish the truth value of a closed formula, because of the need to check for a possibly infinite number of alternative terms to replace for quantified variables. This makes the notion of implication less straightforward than in propositional logic.

In outline, defining implication requires a *semantic model* of the constants, functions, and predicates considered in a particular knowledge base. Formally, the semantic model $\mathcal{M}$ of a given universe $\mathcal{C}$ of logical constants, a set $\mathcal{F}$ of logical functions, and a set $\mathcal{P}$ of logical predicates (these are just specific instantiations of the formal expressions generated as described in Table 5) has three types of elements:

- A non-empty set $\mathcal{U}$, the *universe of concrete values*.

- For each function f in $\mathcal{F}$ with n arguments, a (possibly infinite) list of $n + 1$-tuples

$$(a_1, \ldots, a_n, f_n)$$

of elements in $\mathcal{U}$. This list is called the *concrete function* corresponding to f.

- For each predicate P in $\mathcal{P}$ with n arguments, a (possibly infinite) list of n-tuples

$$(b_1, \ldots, b_n)$$

of elements in $\mathcal{U}$. This list is called the *concrete predicate* corresponding to P.

Intuitively, having a model of these items means "knowing what we are talking about." For instance, if we build a knowledge base in predicate logic to talk about integer arithmetic, the universe $\mathcal{U}$ of concrete values is the set of integers. The functions in our knowledge base are not just formal symbols, but they represent arithmetic operations. For instance, we might introduce a function $f(x, y)$ that given two integers x and y returns their sum. Then, the concrete function corresponding to f is the following infinite list

$$
\begin{array}{rrr}
(0, & 0, & 0) \\
(0, & 1, & 1) \\
(0, & -1, & -1) \\
(1, & 0, & 1) \\
(-1, & 0, & -1) \\
(0, & 2, & 2) \\
(0, & -2, & -2) \\
(1, & 1, & 2) \\
(1, & -1, & 0) \\
(-1, & 1, & 0) \\
(-1, & -1, & -2) \\
(2, & 0, & 2) \\
(-2, & 0, & -2) \\
(0, & 3, & 3) \\
& \vdots &
\end{array}
$$

This sequence merely lists all pairs of integers with their sums. It is an instructive aside to see how it is generated. Of course, the third element of each triple is merely the sum of the first two elements, so we only focus attention on the latter. First, generate all pairs of whole numbers by a very useful procedure called *dovetailing*:

```
for s = 0 to ∞:
    for a = 0 to s
        write (a, s − a)
    end
end
```

This procedure would generate the pairs $(0,0), (0,1), (1,0), (0,2), (1,1), (2,0), (3,0), \ldots$ Look for these pairs in the longer sequence given earlier.

Then between each such pair and the next, we insert all sign variations for that pair. For instance, between $(1,1)$ and $(2,0)$ we insert the following sign variations of $(1,1)$: $(1,-1)$, $(-1,1)$, $(-1,-1)$. Check that this procedure gives the pairs in the first two columns of the list of triples above.

Finally, the third element in each triple is the sum of the first two.

This dovetailing procedure was described in detail simply because it is a useful one to know in discrete mathematics. Coming back to our semantic models, the list of triples given above specifies the meaning of integer sum by listing all possible addends and their sums. The corresponding formal function $f(x, y)$ is assumed to "work" like the concrete function, in the sense that for instance the term $f(1, 1)$ can always be replaced with the constant 2 in any formula in which it appears.

Suppose now that our knowledge base contains the predicate $P(x, y)$ with the following concrete predicate:

$$(\quad 0, \quad 0)$$
$$(\quad 1, \quad 1)$$
$$(-1, -1)$$
$$(\quad 2, \quad 2)$$
$$(-2, -2)$$
$$(\quad 3, \quad 3)$$
$$(-3, -3)$$
$$\vdots$$

(no dovetailing here). Then it should be easy to recognize $P(x, y)$ as the formal equivalent of equality, the "concrete meaning" of this predicate.

We are now ready to sketch the notion of implication in predicate logic. Let Φ be a set of formulas, and ψ be another formula. Each of the formulas involved can be converted into a statement in the semantic model by replacing each constant with its corresponding concrete object in the universe, and each function value with the value we can look up in the corresponding concrete function. For each predicate and list of arguments for it, we can check whether the predicate is true or false for that list of arguments by verifying whether the list of arguments appears in the concrete table for that predicate (the predicate evaluates to true if the list is found, and to false if it is not). Truth values for combinations of the resulting truth values through logical connectives can be determined by using the truth tables of the connectives. Finally, the truth values of quantified predicates can be checked by replacing in turn the variables with all possible objects in the universe, and determining whether the predicate is true for all (in the case of $\forall$) or some (in the case of $\exists$) n-tuples of concrete objects.

In other words, we have outlined a laborious, often infinitely long procedure for translating logical formulas into statements about concrete facts, and for verifying whether the formulas are true or false in the model. Logicians are not concerned that this procedure might not be executable

in a finite amount of time. The translation from predicate logic to a model is merely a conceptual construction to give meaning to the notion of truth. In propositional logic, this notion was specified by a collection of truth tables. The concept of a model is meant to replace that notion in the more difficult scenario of predicate logic.

Here, finally, is the definition of implication. The formulas Φ are said to *imply* the formula ψ in the model $\mathcal{M}$,

$$\Phi \models_{\mathcal{M}} \psi$$

if the translation of ψ in $\mathcal{M}$ is true whenever the translations of all the formulas of Ψ in $\mathcal{M}$ are true.

A more straightforward way to say all of this is to say that Φ implies ψ if whatever ψ means is true whenever whatever Ψ means is true. This requires us to have different, extra-logical methods to determine the truth of statements that we can also express in predicate logic.

Some implication relationships hold in all possible models. In these cases, the $\mathcal{M}$ subscript is dropped from the implicationsymbol. For instance,

$$\phi, \; \phi \rightarrow \psi \; \Rightarrow \; \psi$$

states that the preconditions for applying the Modus Ponens implythe new formula that this rule adds to the knowledge base. This implication is *structural*, in the sense that it is a consequence of the definition of conditional, not of the particular meaning of the formulas ϕ and ψ. No matter how you build a model, this implication relation cannot fail to hold.

In addition, some formulas hold no matter what. We have already encountered these *tautologies* in propositional logic. An example of a tautology in predicate logic is

$$\forall x \left(P(x) \lor \sim P(x) \right),$$

an enriched version of a similar tautology in propositional logic. If formula ϕ is a tautology we write

$$\mathbb{T} \; \Rightarrow \; \phi$$

(truth implies ϕ) or even more simply

$$\Rightarrow \; \phi \; .$$

All these considerations on models, truth, and implication are meta-logical. They are considerations *we* make towards finding out if a particular logical inference system is sound and complete, that is, ultimately useful. Strictly speaking, as was said earlier, logic itself is not concerned with models or meaning.

7 Resolution

The inference rule of Modus Ponens used as an example throughout the previous Sections is an instance of what logicians call *natural* inference rules, because this rule reflects a common argument format for humans. For instance, it is common for us to argue that if there is a lightning bolt, and if every lightning bolt is followed by a thunder clap, then there is a clap of thunder. Note that

we are not stating that if there is a lightning bolt then there is a clap of thunder, but only that this is the case *if* we also assume that thunder bolts always follow lightning bolts. This is the line of reasoning of the Modus Ponens.

Logicians have developed a variety of natural inference systems. These are psychologically useful because they are "user friendly," and they may have philosophical value in that they may describe certain aspects of human thought.

For computer use, on the other hand, natural inference systems are hard to use. The large number of inference rules they contain poses the problem of choosing the appropriate rules to apply every step of the way. Without a human in the machine to guide the choice, this is a very difficult problem.

The inference system called *resolution* was developed in 1965 by the British-born logician and philosopher John Alan Robinson while at Rice University in Houston, Texas. His aim was to devise a systematic inference system that could be programmed on a computer. His resolution algorithm is the basis of a logic programming language called *Prolog*.

Resolution is applicable to formulas that are written in a standardized format called *Conjunctive Normal Form* (*CNF*). Once the knowledge base and the formula to be proven are written in CNF, resolution uses a single inference rule (not surprisingly called *resolution* itself). Of course, there is still the question as to which formulas in the knowledge base the rule should be applied to in which order. In addition, given a particular set of formulas, resolution can be applied in many ways, and the computer program must make a choice here as well. These choices are not trivial.

Even so, resolution can be proven to be both sound and complete, as it can prove all (completeness) and only (soundness) the predicates that the knowledge base implies.

The next Section discusses the conjunctive normal form. Section 7.2 shows the resolution rule and an example of its application, and proves soundness. Completeness will not be proven here.

7.1   Conjunctive Normal Form

A formula of predicate logic is said to be in *Conjunctive Normal Form* (*CNF*) when it is consistent with the BNF grammar in Table 6.

Thus, a formula in CNF is a universally quantified conjunction of disjunctions, that is, an "and" of several "or"s of *literals* (predicates or their negations). This may be a bit confusing: each sub-formula of a conjunction (the "and" of several sub-formulas) is called a conjunct (because it is con-joined with other sub-formulas), but each sub-formula is also called a disjunction, because it disjoins (is the "or" of) several literals. The production for $<conjunct>$ in the Table should make this clear.

Here is an example of a formula in CNF:

$$\forall x \, \forall y \, ((\sim P(x,y) \vee R(x,f(x))) \wedge (\sim Q(x,y) \vee R(x,f(x))))$$

Since only closed formulas are allowed and there are no existential quantifiers in CNF, all variables are always universally quantified, so the quantifiers themselves are redundant. Because of this, it is common to drop all quantifiers and the parentheses around $<conjunction>$. In addition, the $\wedge$ symbols are usually replaced by commas, which in turn allows dropping the parentheses

<cnf> ::= *<conjunction>* | *<quantifiers>* (*<conjunction>*)

<conjunction> ::= *<conjunct>* | *<conjunct>* ∧ *<conjunction>*
<conjunct> ::= (*<disjunction>*)
<disjunction> ::= *<literal>* | *<literal>* ∨ *<disjunction>*
<literal> ::= *<predicate>* | ∼ *<predicate>*

<quantifiers> ::= *<quantifier>* | *<quantifier>* *<quantifiers>*
<quantifier> ::= ∀ *<variable>*

The definitions of *<predicate>* and *<variable>* are as in Table 5.
In addition to being consistent with the grammar above, a CNF formula must satisfy the following requirements:

- It must be a *closed formula*, so no free variables are allowed.

- Variables associated to different (universal) quantifiers must have different names, that is, they must be *standardized apart*.

Table 6: Conjunctive Normal Form, CNF.

<cnf> ::= *<disjunction>* | *<disjunction>* , *<cnf>*
<disjunction> ::= *<literal>* | *<literal>* ∨ *<disjunction>*
<literal> ::= *<predicate>* | ∼ *<predicate>*

The definition of *<predicate>* is as in Table 5.
In addition to being consistent with the grammar above, an (abbreviated) CNF formula must satisfy the following requirements:

- It must be a *closed formula*, so all variables are assumed to be universally quantified.

- Variables associated to different implicit (universal) quantifiers must have different names, that is, they must be *standardized apart*.

Table 7: Abbreviated Conjunctive Normal Form.

around <*disjunction*>. This yields the abbreviated CNF in Table 7. From now on, we will use the abbreviated form. The example above is then rewritten as follows:

$$\sim P(x,y) \lor R(x, f(x)), \ \sim Q(x,y) \lor R(x, f(x)).$$

Any formula in predicate logic can be converted to CNF through a sequence of steps. To illustrate what each step does, we consider as an example the following sentence:

> Anyone who either owns or rents a cell phone has a service agreement.

To formalize, suppose that our universe of objects is a group of people, a list of cell phones, and a catalog of service agreements. Define the following predicates:

$P(x,y)$ means "x owns cell phone y."

$Q(x,y)$ means "x rents cell phone y."

$R(x,y)$ means "x has service agreement y."

Then, the sentence above can be written as follows:

$$\forall x \ ((\exists y \ (P(x,y) \lor Q(x,y))) \ \rightarrow \ (\exists y \ R(x,y)))$$

and this reads

> For every person x, if there is a cell phone y such that x either owns or rents y, then there is a y such that x has service agreement y.

This is not the only formalization possible for this sentence. A more general translation would introduce a predicate $C(y)$ for "y is a cell phone," and another predicate $O(x,y)$ for "x owns y." Then, "x owns cell phone y" would translate to $C(y) \land O(x,y)$. This solution is more flexible, because it separates the notion of ownership from what is being owned. However, we stick to the simpler formalization as an example.

The use of y in both the premise and the conclusion of the conditional above is purposely ambiguous in English, to make the point that in logic this use is not ambiguous at all. This is because the y that appears twice in $P(x,y) \lor Q(x,y)$ is in the scope of an existential quantifier that is different from the existential quantifier whose scope contains the y in $R(x,y)$. These two scopes are disjoint (that is, they are not nested within each other), so these occurrences of y can refer to different objects. When using resolution, this would cause problems, hence the requirement that quantified variables be standardized apart, that is, given different names. The *standardize apart* step in the procedure below enforces this requirement.

Skolemization is a simple but important transformation used to remove every occurrence of existential quantifiers from a formula. This transformation replaces the variables in the scope of an existential quantifier with either a *Skolem*[9] *constant* or a *Skolem function*, and then removes the existential quantifier. Let us see how this works.

---

[9] After the Norwegian logician Thoralf Albert Skolem, 1887–1963.

To say that $\exists x\, P(x)$ means that somewhere in the universe of objects there are objects for which the predicate P holds. So it is OK to introduce a fresh constant, say c_{213}, and use it to denote one of those objects. The closed predicate $P(c_{213})$ is now equivalent to the existentially quantified formula, with the object c_{213} testifying, so to speak, to the existence of at least one object for which the predicate P holds. We do not need to know what that object denotes. We do know that one exists, because the closed formula $\exists x\, P(x)$ so states.

However, skolemization by a constant is *not* sound when the existential quantifier is nested within the scope of one or more universal quantifiers. Consider for instance the sentence

<p style="text-align:center">Everyone has a cell phone.</p>

With our running definitions, this formalizes to the following closed formula:

$$\forall x\, \exists y\, P(x, y) \ .$$

If we skolemize this formula with a constant, we obtain

$$\forall x\, P(x, c)$$

which reads "Everyone has cell phone c." This is not what the original formula meant: That one stated that for each person there is a cell phone owned by that person. The cell phones of different people may be (although do not have to be) different ones. The skolemized version, on the other hand, states that everyone owns the *same* phone, denoted by the constant c.

To avoid this, the existentially quantified variable is replaced by a function (named with a fresh name) whose arguments are all the universally quantified variables whose scopes contain the existential quantifier being eliminated. In the cell phone example, function skolemization yields the following:

$$\forall x\, P(x, f(x))$$

where f is a name that does not appear anywhere else. Just as for constants, one can always make up new function names in predicate logic (see the production for `function-name` in Table 5).

This works correctly, because for every x the function $f(x)$ denotes that person's individual phone (which could of course happen to coincide with someone else's), and phone sharing is no longer compulsory.

As a more elaborate example, skolemization of

$$\forall x\, \forall y\, \exists z\, \forall w\ (P(x, f(y, z)) \wedge Q(z, w))$$

yields

$$\forall x\, \forall y\, \forall w\ (P(x, f(y, g(x, y))) \wedge Q(g(x, y), w)) \ .$$

We are now ready to describe the procedure for **conversion to CNF:**

Eliminate $\rightarrow$ using the logical equivalence

$$\phi \rightarrow \psi \equiv\ \sim \phi \vee \psi \ .$$

The example becomes

$$\forall x\ (\sim (\exists y\ (P(x, y) \vee Q(x, y)))\ \vee\ (\exists y\, R(x, y)))\ .$$

Move $\sim$ inward as far as possible using De Morgan's laws for $\vee$, $\wedge$, $\forall$, $\exists$:

$$\begin{aligned}
\sim (\phi \vee \psi) &\equiv \sim \phi \wedge \sim \psi \\
\sim (\phi \wedge \psi) &\equiv \sim \phi \vee \sim \psi \\
\sim \forall x\, \phi &\equiv \exists x \sim \phi \\
\sim \exists x\, \phi &\equiv \forall x \sim \phi \, .
\end{aligned}$$

The example becomes

$$\forall x\ ((\forall y\ (\sim P(x,y) \wedge \sim Q(x,y))) \vee (\exists y\, R(x,y)))\ .$$

Standardize apart variables by renaming homonyms in different quantifier scopes. Recall that predicate logic provides an infinite supply of variable names. Same variable names within the scope of the same quantifier remain the same.

The example becomes

$$\forall x\ ((\forall y\ (\sim P(x,y) \wedge \sim Q(x,y))) \vee (\exists z\, R(x,z)))\ .$$

Note the two occurrences of y in the scope of the first existential quantifier, and the *same* variable x in the three predicates.

Move quantifiers left to the beginning of the formula. Since variables are standardized apart, the formula resulting from this move left is equivalent to the original. This should be obvious, but a formal proof would be tedious. The resulting formula is said to be in *prenex* form.

The example becomes

$$\forall x\, \forall y\, \exists z\ ((\sim P(x,y) \wedge \sim Q(x,y)) \vee R(x,z))\ .$$

Skolemize all existentially quantified variables.

The example becomes

$$\forall x\, \forall y\ ((\sim P(x,y) \wedge \sim Q(x,y)) \vee R(x,f(x,y)))\ .$$

Distribute $\vee$ over $\wedge$ using the logical equivalence

$$\phi \vee (\psi \wedge \omega) \equiv (\phi \vee \psi) \wedge (\phi \vee \omega)$$

and its version obtained by applying the commutative property of $\vee$.

The example becomes

$$\forall x\, \forall y\ ((\sim P(x,y) \vee R(x,f(x,y))) \wedge (\sim Q(x,y) \vee R(x,f(x,y))))\ .$$

This completes the logical part of conversion to CNF. The abbreviated form can then be obtained by removing all unnecessary parentheses, dropping all universal quantifiers, and changing $\wedge$ symbols to commas.

The example becomes

$$\sim P(x,y) \vee R(x,f(x,y))\, ,\ \sim Q(x,y) \vee R(x,f(x,y))\ .$$

7.2 The Resolution Rule

The single rule that resolution uses for inference uses the two technical notions of substitution and unification, which are explained next.

Substitution. If ϕ is a literal in abbreviated CNF, t is a term, and x is a variable (these objects are defined by the BNF grammars of Table 5 and Table 6), then applying the *substitution*

$$t/x$$

to ϕ results into

$$\text{SUB}(t/x, \phi) \,,$$

which is defined as the new predicate obtained by substituting every instance of x in ϕ with t.

This rule takes proper care of quantification when it is applied to literals in the abbreviated CNF of Table 7. Without abbreviation (that is, if universal quantifiers are explicit), there are problems. For instance,

$$\text{SUB}(f(x,y)/z, \forall z\, P(z)) \equiv \forall f(x,y)\, P(f(x,y)) \,.$$

The result is no longer a formula in predicate logic, because we can only quantify on variables, not on functions of variables (see the last alternative in the production for *<formula>* in Table 5). What we really would like the substitution to do is to transform the original literal into the following:

$$\forall x\, \forall y\, P(f(x,y)) \,.$$

However, this is exactly the result when we use the abbreviated form:

$$\text{SUB}(f(x,y)/z, P(z)) \equiv P(f(x,y)) \,,$$

since the last expression in abbreviated form implicitly universally quantifies on all free variables, that is it quantifies on x and y.

Similarly, substitution of a variable with a ground term makes the (implicit) quantifier disappear:

$$\text{SUB}(a/x, P(x)) \equiv P(a) \,.$$

Substitution for a variable that does not exist in ϕ does nothing. In that case, $\text{SUB}(t/x, \phi) \equiv \phi$. For instance,

$$\text{SUB}(f(x)/y, P(x)) \equiv P(x) \,,$$

since the formula $P(x)$ does not contain the variable y.

Unification. Let ϕ, ψ be two literals in abbreviated CNF. Then the *unifier*

$$\sigma = \text{UNI}(\phi, \psi)$$

is the most general list σ of substitutions that make ϕ and ψ become logically equivalent:

$$\text{SUB}(\sigma, \phi) \equiv \text{SUB}(\sigma, \psi) \,.$$

In this context, "most general" means that the substitution replaces as few variables as possible. Also, applying a list

$$\sigma = \{s_1, \ldots, s_k\}$$

of substitutions means applying them all in turn, in the order in which they are listed:

$$\text{SUB}(\sigma, \phi) \equiv \text{SUB}(s_k, \text{SUB}(\ldots \text{SUB}(s_1, \phi) \ldots)) .$$

For instance,

$$\text{UNI}(\sim P(a, x), \sim Q(a, y)) = \{\} .$$

(note the use of equality rather than logical equivalence, $\equiv$: the two sides are not formulas). The two braces on the right-hand side denote the empty list. This unification fails because there is no way to unify the two literals, so the resulting list of substitutions is empty.

Another example:

$$\text{UNI}(P(a, x), P(y, f(y))) = \{y/a, x/f(a)\} .$$

This is a bit trickier to find, but it is easy to verify that the two substitutions listed in braces transform both literals into $P(a, f(a))$.

Resolution. We are now ready to state the *resolution inference rule*.

Let

$$p_1 \vee \ldots \vee p_m \quad \text{and} \quad q_1 \vee \ldots \vee q_n$$

be two conjuncts (that is, two disjunctions) in an abbreviated CNF formula, and let

$$\sigma = \text{UNI}(p_1, \sim q_1)$$

be a nonempty list of substitution that unify p_1 and the negation of q_1.
Then the following conjunct, called the *resolvent*, can be added to the knowledge base:

$$\text{SUB}(\sigma, p_2 \vee \ldots \vee p_m \vee q_2 \vee \ldots \vee q_n) .$$

Resolution is a complete proof system when it is used with *refutation*. That is, to prove the formula ϕ, add $\sim \phi$ (translated to abbreviated CNF) to the knowledge base, and prove $\mathbb{F}$, a contradiction.

In this statement of the resolution inference rule, the *first* literals of the two conjuncts are resolved away (note that p_1 and q_1 have both disappeared from the resolvent). Since disjunction is commutative, this rule allows resolving away *any* two literals with the proper format: first bring them to the front of their disjunctions, then apply the resolution rule.

Refutation is also known as *proof by contradiction* or *reduction ad absurdum*. Some formulas may be provable by resolution directly, that is, without using refutation, but not all of them can.

All formulas that the knowledge base implies can be proven by resolution with refutation. For a proof of this completeness result see for instance the original paper, J. A. Robinson, "A Machine-Oriented Logic Based on the Resolution Principle," *Journal of the Association for Computing Machinery*, 12(1):23–41, 1965.

Soundness.   The proof of soundness for resolution is straightforward: The conjunct

$$p_1 \vee \ldots \vee p_m \,, \ q_1 \vee \ldots \vee q_n$$

is a universal statement, in the sense that it holds for all values of whatever variables appear in the literals $p_1, \ldots p_m, q_1, \ldots q_n$ (recall that all variables in an abbreviated CNF are implicitly universally quantified). So if we replace some of the variables by special values through the substitution σ, the conjunct above is still true: if everything satisfies some property, then any particular object also satisfies that property.

As a consequence, each of the two formulas

$$\phi \equiv \text{SUB}(\sigma, p_1 \vee \ldots \vee p_m) \quad \text{and} \quad \psi \equiv \text{SUB}(\sigma, q_1 \vee \ldots \vee q_n)$$

is separately true. If we let

$$\lambda \equiv \text{SUB}(\sigma, p_1) \,,$$

then since the substitution σ was found so as to make $\text{SUB}(\sigma, p_1)$ and $\sim \text{SUB}(\sigma, q_1)$ logically equivalent, we also have

$$\sim \lambda \equiv \text{SUB}(\sigma, q_1)$$

so that

$$\phi \equiv \lambda \vee \alpha \quad \text{and} \quad \psi \equiv \ \sim \lambda \vee \beta$$

where we defined

$$\alpha \equiv \text{SUB}(\sigma, p_2 \vee \ldots \vee p_m) \quad \text{and} \quad \beta \equiv \text{SUB}(\sigma, q_2 \vee \ldots \vee q_n) \,.$$

With these definitions, the resolvent is equal to

$$\alpha \vee \beta$$

(check this), and this must be true, as we now show.

The literal λ is either true or false. Suppose that it is true. Then since $\psi = \sim \lambda \vee \beta$ is true and $\sim \lambda$ is false, β must be true, so in this case the resolvent is true. Suppose now that λ is false. Then since $\phi = \lambda \vee \alpha$ is true, α must be true, and the resolvent is therefore true in this case as well.

In conclusion, the two original conjuncts imply the resolvent, and the inference rule is sound.

Examples. Let us consider first a simple, "regular" example of resolution to understand the basic rule. The conjuncts

$$\sim P(x) \vee Q(x), \; P(a) \vee R(x)$$

are candidates for resolution, because the literal P appears without negation in the first conjunct, and with negation in the second. The following substitution unifies these two literals:

$$\text{UNI}(\sim P(x), \sim P(a)) \; = \; \{x/a\} \; .$$

To apply the resolution rule, we form a new conjunct, the resolvent, by taking the "or" (disjunction) of the two old conjuncts, dropping the literals that have been unified, and applying the substitution x/a. This yields the new disjunction

$$Q(a) \vee R(a) \; ,$$

which can be added as a new conjunct to the knowledge base:

$$\sim P(x) \vee Q(x), \; P(a) \vee R(x), \; Q(a) \vee R(a) \; .$$

In a somewhat degenerate case, one of the two conjuncts being resolved is a single literal. Consider for instance the conjunction

$$\sim P(x) \vee Q(x), \; P(a) \; .$$

The unifier is again

$$\text{UNI}(\sim P(x), \sim P(a)) \; = \; \{x/a\}$$

and after dropping the unified literals $\sim P(a)$ and $P(a)$ the resolvent is

$$Q(a) \; .$$

Let us take this one step further, and consider resolving the conjunction

$$\sim P(x), \; P(a) \; .$$

The unifier is still

$$\text{UNI}(\sim P(x), \sim P(a)) \; = \; \{x/a\}$$

and nothing is left after dropping the unified literals $\sim P(a)$ and $P(a)$. What does "nothing" mean?

After substitution, but before the unified literals are dropped, the conjunct reads

$$\sim P(a), \; P(a)$$

where the comma means "and." This formula is false, since $P(a)$ cannot be simultaneously true and false. To preserve soundness, the resolvent must be $\mathbb{F}$, a contradiction.

As a more complex example, Section 7.1 showed that the sentence

> Anyone who either owns or rents a cell phone has a service agreement.

can be formalized as follows

$$\forall x \ ((\exists y \ (P(x, y) \lor Q(x, y))) \ \rightarrow \ (\exists y \, R(x, y)))$$

and translated to the following abbreviated-CNF formula:

$$\sim \ P(x, y) \lor R(x, f(x, y)) \, , \ \sim \ Q(x, y) \lor R(x, f(x, y)) \ .$$

Let us add the following piece of information to the knowledge base:

Alan does not have a service agreement and does not own a cell phone.

Formally,

$$\sim \ \exists y \, P(a, y) \land \sim \ \exists y \, R(a, y)$$

and in abbreviated CNF

$$\sim P(a, y) \, , \ \sim R(a, z)$$

(notice standardization). Thus, the entire knowledge base is

$$\sim \ P(x, y) \lor R(x, f(x, y)) \, , \ \sim \ Q(x, y) \lor R(x, f(x, y)) \, , \ \sim P(a, z) \, , \ \sim R(a, u)$$

where the constant a denotes Alan, and where variables were further standardized apart.

Let us now use resolution with refutation to prove that

Alan does not not rent a cell phone

that is,

$$\sim Q(a, y) \ .$$

To this end, we negate the theorem to be proven, standardize its variable apart from those already in the knowledge base, and add it to the known facts to obtain the following conjunction of five conjuncts:

$$\sim \ P(x, y) \lor R(x, f(x, y)) \, , \ \sim \ Q(x, y) \lor R(x, f(x, y)) \, , \ \sim P(a, z) \, , \ \sim R(a, u) \, , \ Q(a, v) \ .$$

To apply resolution, we need to find literals that appear once with and once without a negation symbol. Both Q and R fit the bill. Let us choose (arbitrarily) the unification:

$$\text{UNI}(R(x, f(x, y)), R(a, u)) \ = \ \{x/a, \ u/f(a, y))\} \ .$$

with the pair of conjuncts

$$\sim \ Q(x, y) \lor R(x, f(x, y)) \quad \text{and} \quad \sim R(a, u) \ .$$

The resolution rule allows adding to the knowledge base the new disjunction formed from the two conjuncts obtained by dropping the unified literals:

$$\sim Q(a, y) \, ,$$

so the new knowledge base is the following:

$$\sim P(x,y) \vee R(x, f(x,y)), \ \sim Q(x,y) \vee R(x, f(x,y)), \ \sim P(a,z), \ \sim R(a,u), \ Q(a,v), \ \sim Q(a,y).$$

At this point it is clear what to do, since the conjuncts

$$Q(a,v) \quad \text{and} \quad \sim Q(a,y)$$

are just a substitution away from a contradiction:

$$\text{UNI}(Q(a,v), \sim Q(a,y)) \ = \ \{v/y\}$$

(or viceversa, $\{y/v\}$) and the resolvent is $\mathbb{F}$, a contradiction.

In conclusion, the formula $Q(a,v)$ we added to the database leads to a contradiction and must therefore be false. Its opposite, $\sim Q(a,y)$ must be true: Alan does not rent a cell phone.

The resolution procedure, even in this simple example requires making choices, mainly which literals in which conjuncts to resolve. A particular resolution step might lead to a dead end. Try for instance to resolve the pair of conjuncts

$$\sim P(x,y) \vee R(x, f(x,y)) \quad \text{and} \quad \sim R(a,u)$$

as the first step in the example above: not much of use comes out of this step. When proving complex formulas given a large knowledge base, the number of possible alternatives escalates.

In addition, a resolvent may be longer than either conjunct being resolved, so the formulas added to the knowledge base may increase in length. How do we know that some of the formulas generated by this procedure will eventually shrink down to $\mathbb{F}$, a very short formula indeed?

Completeness guarantees that if the initial knowledge base impliesthe formula being proven, then resolution, if it tries all possibilities along the way, will eventually succeed. Efficiency is of course a different story. Logicians have developed heuristics that make resolution inference efficient in many cases, so much so that Prolog, a logical programming language based on resolution, has encountered quite a bit of practical success.

8   What Logic Can and Cannot Do

Let us review and expand on the results about logical proofs from the previous Sections.

Model.   In predicate logic, a *model* is an interpretation of all predicates, functions, and constants, that is, a detailed mapping between the logical symbols one defines and what they mean. We saw in Section 6.2 what form this map takes. In propositional logic, there are no functions or constants, the only predicates are atomic propositions, and all one can do with these is to declare them to be true or false. So in propositional logic a model is a list of the truth value of each atomic proposition used.

Knowledge base.   A knowledge base Φ can contain any type of formula, including atomic propositions. So it is unclear whether a proposition p that is taken to be true should be placed in the knowledge base, or whether its truth value should be recorded as part of the model instead. For propositional logic, this confusion makes the notion of a model unnecessary, since a model would only contain a list of all propositions that are taken to be true. Instead, these propositions can be placed equivalently into the knowledge base. For predicate logic, on the other hand, the distinction between knowledge base and model is useful, because the knowledge base can only include closed formulas, not functions, constants, or open predicates: the meaning of these is part of the model.

To remove this confusion, we now stipulate that *a knowledge base cannot contain atomic propositions*. This restores the usefulness of the model concept for propositional logic, and separates model from knowledge base in both propositional and predicate logic. In both these realms, the model captures the meaning of the individual symbols (and their truth values, when applicable) used in the language. The knowledge base, on the other hand, lists the assumptions about the world that are stated in the form of *combinations* of the basic symbols. The model is more of a dictionary (meaning of words), and the knowledge base is more of an encyclopedia (facts about the world).

Implication within a Model.   The knowledge base Φ *implies* the formula ψ within model $\mathcal{M}$,

$$\Phi \models_{\mathcal{M}} \psi$$

if whatever ψ means within the model $\mathcal{M}$ is true whenever whatever Φ means within the model $\mathcal{M}$ is true. For instance, let

$$\Phi = \{p \vee q\}\,,$$

a very simple knowledge base, and let

$$\psi \equiv p \rightarrow q\,.$$

Then,

$$\Phi \models_{\mathcal{M}_1} \psi$$

where

$$\mathcal{M}_1 = \{p\}$$

(we just list the true propositions in a model. Unlisted propositions are taken to be false.) However, Φ does not implyψ in the model

$$\mathcal{M}_2 = \{q\}\,.$$

Verify the statements made in this and the following examples.

In predicate logic, checking implication involves an appeal to proof methods beyond those in logic: for instance, mathematical proof methods. In propositional logic, which we know to be a subset of predicate logic, checking implication amounts to compiling possibly impractically large truth tables.

Implication *tout court*. A knowledge base Φ may imply some formula ψ within every possible model. The knowledge base Φ is then simply said to *imply ψ*,

$$\Phi \Rightarrow \psi .$$

For instance, the knowledge base

$$\Phi = \{p \wedge q\} ,$$

implies

$$\psi \equiv p \rightarrow q .$$

in all cases, regardless of the truth values of p and q.

Validity. A formula that is true in all knowledge bases is said to be *valid*, or a *tautology*[10]

$$\Rightarrow \psi ,$$

a notation that emphasizes that validity amounts to implication for any knowledge base. For instance,

$$\psi \equiv p \vee \sim p$$

is a tautology.

In principle, one could also define the notion of a formula that is true in a particular model, but otherwise requires no other assumption:

$$\models_{\mathcal{M}} \psi .$$

For instance, if

$$\psi \equiv p \rightarrow q ,$$

we have

$$\models_{\mathcal{M}_1} \psi \quad \text{and} \quad \models_{\mathcal{M}_2} \psi$$

in both models

$$\mathcal{M}_1 = \{q\} \quad \text{and} \quad \mathcal{M}_2 = \{\}$$

($\mathcal{M}_2$ is the model in which both p and q are false), but ψ is not valid in the model

$$\mathcal{M}_3 = \{p\} .$$

However, this last notion of "model-dependent validity" is not useful for the considerations that follow.

---

[10]For technical reasons, some authors introduce a notion of tautology that is stronger than what is used here, and different from that of a valid formula. This distinction is not useful in this context.

Provability.   Let $\mathcal{P}$ be an inference (or proof) system. A formula ψ is *provable* in $\mathcal{P}$ from the knowledge base Φ if it is possible to write a proof for it that uses the rules in $\mathcal{P}$ starting from formulas in Φ. We then write

$$\Phi \vdash_{\mathcal{P}} \psi \; .$$

Soundness.   Given the definitions above, a logic is *sound* if

$$\exists \mathcal{P} \, \forall \Phi \, \forall \psi \, (\Phi \vdash_{\mathcal{P}} \psi \; \rightarrow \; \Phi \Rightarrow \psi) \; .$$

Completeness.   A logic is *complete* if

$$\exists \mathcal{P} \, \forall \Phi \, \forall \psi \, (\Phi \Rightarrow \psi \; \rightarrow \; \Phi \vdash_{\mathcal{P}} \psi) \; .$$

Dissect the last two definitions, and make sure you understand what they state.

8.1   Completeness of Predicate Logic

Kurt Gödel proved in 1930 that *predicate logic is sound and complete.*

Of course, this result holds also propositional logic, a subset of predicate logic. Soundness merely states that one can find a proof system whose inference rules make sense: if their preconditions are true, they will not produce false formulas. The statement that predicate logic is both sound and complete, on the other hand, is rather strong, because completeness means that there exist all-powerful proof systems for predicate logic: A formula that is entailed by some (any!) knowledge base can *always* be proven from it with the given inference rules.

A consequence of Gödel's result is that implication and provability, two concepts very different in their definitions, are nonetheless equivalent to each other. Robinson's resolution system (Section 7) made Gödel's result constructive, by actually exhibiting a proof system with the required properties.

8.2   Semi-Decidability of Predicate Logic

Gödel's completeness theorem would seem to solve all our problems in predicate logic, at least if we are patient enough. Given a set Φ of axioms, we could conceivably write a program that applies the resolution inference rule to all formulas in Φ in turn, and in all possible ways. The program could then add the results of this first round to the knowledge base, and repeat, thereby generating all possible theorems that follow from Φ.

However, there are two problems with this idea. First, this would take an infinite amount of time, because there is an infinite number of theorems. However, even for a single theorem, we don't know how long to wait. We cannot ask the program to output theorems in order, say, of increasing length: the fact that theorem 1 is shorter than theorem 2 does not mean that theorem 1 needs less work to be proven. So for any particular theorem, we cannot predict how long the program would take to output it: eventually it will come out, but there is no systematic way to

bound the time we have to wait. We may be able to find a time bound for *some* theorems, but not a procedure that bounds the time for *any* theorem.

The second problem is more fundamental, and arises when we try to prove a formula ψ for which we do not know whether it is entailed by Φ. Naïvely, we could run two versions of the same program in parallel: the first tries to prove ψ, the second tries to prove $\sim \psi$. If one of them is entailed by Φ, it will eventually be produced by the program that attempts to prove it.

This argument, however, is flawed: if Φ does not imply ψ, then it does not necessarily imply $\sim \psi$, either. This is because ψ is either true or false. If it is true it may or may not be entailed by Φ. Similarly, if ψ is false, its negation $\sim \psi$ may or may not be entailed by Φ.

Suppose then that we run our two programs, one trying to prove ψ and the other $\sim \psi$. It is still possible, in spite of Gödel's completeness result, that neither program will ever produce a result. Say that after a year both programs are still resolving away without having found a proof. What can we say? Not much. It could be that Φ implies one of the two formulas, and that the corresponding prover will produce a proof tomorrow, or in two more years. Or it could be that neither formula is entailed by the database. In that case, no proof will ever be found (because of soundness). This asymmetric state of affairs is called *semi-decidability*: *if* the knowledge base implies a formula, *then* the theorem prover will eventually find a proof. However, if the knowledge base does not imply a formula, we may not be able to ever settle the question as to whether the formula is or is not entailed by the knowledge base. Alas, this is not a problem with resolution, but with predicate logic itself: predicate logic can be shown to be *semi-decidable*: there is no proof system that determines whether any given formula is entailed by a given knowledge base.

The negative result of semi-decidability does not apply to propositional logic: *propositional logic is decidable*. This is immediate to prove: formulas in propositional logic are finite combinations of a finite number of atomic propositions, so one can in principle build a truth table for any formula ψ to be proven, and settle the question.

8.3 Gödel's Incompleteness Theorem

The name of Kurt Gödel, the Austrian logician who proved the soundness and completeness of predicate logic, is also associated to an even more famous *incompleteness theorem*. This relates to the expressive power of the knowledge base Φ, together with the power of the proof system.

Gödel first observed that no knowledge base in predicate logic can cover all of arithmetic: the language is not expressive enough. However, if predicate logic is extended by just adding something called the *induction schema*, then the new, more powerful language is sufficient to express every known fact of arithmetic. Is this more powerful language still sound and complete?

Gödel's answer is negative: given any such extended knowledge base and any proof system, it is always possible to construct formulas that are entailed by the axioms but cannot be proven from them. That is, these formulas are true within a model of arithmetic, and yet the given proof system cannot prove them from the axioms.

This result, which Gödel achieved in 1931, just a year after proving soundness and completeness of predicate logic, seems to have in a sense shattered the ancient dream of building a machine that thinks: If the language of this machine is predicate logic, it is too simple for talking even

about arithmetic. If the language is extended to make it powerful enough, then not every truth it can express can be proven by the machine.

Whether this negative view is justified is still an open debate, beyond the scope of this brief introduction to logic.