Formal logic

Miguel Palomino

1 Introduction

Logic studies the validity of arguments. A typical case in point is that of syllogisms: logical arguments in which, starting from two premises, a conclusion is reached. For example, given that

> There are horses in Spain.
> All horses are mammals.

it can be inferred that

> There are mammals in Spain.

Of course, if instead of the second premise we had the weaker one

> Some horses are mammals.

where the universal (*all*) has been replaced with an existential (*some*/*exists*) then the argument would not be valid. In Ancient Greece, Aristotle exhaustively considered all possible combinations of universals and existentials in syllogisms, allowing also for the possibility of negations, and collected those corresponding to valid inferences in a classification theorem. For many centuries, that classification (slightly enhanced by the scholastics during the Middle Ages) was all there was to know about logic.

In its origin, the term "formal" logic used to be a reference to the form of the arguments: the validity of an argument depends exclusively on the *form* of the premises and the conclusion, not on whether these are true or false. In the previous example, if we were to replace "horses" with "unicorns" the argument would still be valid, regardless of the fact that unicorns do no exist.

Nowadays, however, "formal" refers to the use of the formal and rigorous methods of mathematics in the study of logic that began to be put into practice in the second half of the 19th century with George Boole and, especially, Gottlob Frege (see [1]). This trend started with a shift to a symbolic notation and artificial languages, and gradually evolved until, in 1933 with Tarski [2], it culminated with the withdrawal from an absolute notion of Truth and instead focused on the particular truths of concrete structures or models.

2 Propositional logic

Arguably, the simplest logic is *propositional logic* and we will use it to introduce the underlying elements of every logic. Given a set $A = \{p, q, r, \ldots\}$ of *atomic propositions*, the language of propositional logic is constructed according to the following rules:

$$\varphi ::= p \in A \mid \neg p \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \varphi \to \varphi \mid \varphi \leftrightarrow \varphi$$

$\neg$ means 'No'	$\to$ means 'Implies/Then'
$\vee$ means 'Or'	$\leftrightarrow$ means 'If and only if'
$\wedge$ means 'And'	

For example, let us assume that p represents "The chef is competent", q represents "The ingredients are expired", and r, "The cake is delicious". Then, the premise "If the chef is competent and the ingredients are not expired, then the cake will be delicious" could be represented in the language of propositional logic as

$$(p \wedge \neg q) \to r\,.$$

Furthermore, if we assume that the chef is actually competent, that is, if we assume p as the second premise, we can conclude that "If the cake is not delicious, the ingredients are expired", or, formally:

$$\neg r \to q\,.$$

But how and why can we conclude that this last sentence follows from the previous two premises? Or, more generally, how can we determine whether a formula φ is a valid consequence of a set of formulas $\{\varphi_1, \ldots, \varphi_n\}$? Modern logic offers two possible ways, that used to be fused in the time of syllogisms: the model-theoretic approach and the proof-theoretic one.

In model theory it is necessary to assign a meaning to the formulas, to define a *semantics* for the language. The central notion is that of *truth* and of deciding the circumstances under which a formula is true. The more complex the logic, the more difficult this assignment is and hence the more complex the semantics. In propositional logic, we have to start by assigning arbitrary values to the atomic propositions: a *valuation* V is defined as a function that maps atomic propositions to either 0 (meaning, intuitively, false) or 1 (true). The meaning $\mathcal{I}^V(\varphi)$ of an arbitrary formula φ is defined recursively:

$$\mathcal{I}^V(p) = V(p)$$

$$\mathcal{I}^V(\neg\varphi) = \begin{cases} 1 & \text{if } \mathcal{I}^V(\varphi) = 0 \\ 0 & \text{if } \mathcal{I}^V(\varphi) = 1 \end{cases}$$

$$\mathcal{I}^V(\varphi \vee \psi) = \begin{cases} 1 & \text{if } \mathcal{I}^V(\varphi) = 1 \text{ or } \mathcal{I}^V(\psi) = 1 \\ 0 & \text{otherwise} \end{cases}$$

$$\mathcal{I}^V(\varphi \wedge \psi) = \begin{cases} 1 & \text{if } \mathcal{I}^V(\varphi) = 1 \text{ and } \mathcal{I}^V(\psi) = 1 \\ 0 & \text{otherwise} \end{cases}$$

$$\mathcal{I}^V(\varphi \to \psi) = \begin{cases} 1 & \text{if } \mathcal{I}^V(\varphi) = 0 \text{ or } \mathcal{I}^V(\psi) = 1 \\ 0 & \text{otherwise} \end{cases}$$

$$\mathcal{I}^V(\varphi \leftrightarrow \psi) = \begin{cases} 1 & \text{if } \mathcal{I}^V(\varphi) = \mathcal{I}^V(\psi) \\ 0 & \text{otherwise} \end{cases}$$

For example, if $V(p) = V(q) = 0$ and $V(r) = 1$, then $\mathcal{I}^V(\neg p) = 1$, $\mathcal{I}^V(\neg p \wedge q) = 0$, and $\mathcal{I}^V(r \to (\neg p \wedge q)) = 0$.

If $\mathcal{I}^V(\varphi) = 1$ then it is said that V is a *model* of φ, or that V *satisfies* φ; it is a "world" in which φ is true. A formula is said to be *valid* if it is true under all circumstances, that is, if every valuation is a model of φ:

$$\varphi \text{ is valid if } \mathcal{I}^V(\varphi) = 1 \text{ for all valuations } V\,.$$

For instance, it is easy to check that $p \to (q \to p)$ is a valid formula. Similarly, if V is a model of all the formulas in a set Γ then V is said to be a model of Γ. A formula φ is a *semantic consequence* of a set Γ of formulas, written $\Gamma \models \varphi$, if every model of Γ is also a model of φ or, alternatively, if φ is true whenever all formulas in Γ are true:

$$\Gamma \models \varphi \text{ if } \mathcal{I}^V(\varphi) = 1 \text{ whenever } \mathcal{I}^V(\psi) = 1 \text{ for all } \psi \in \Gamma\,.$$

In the proof-theoretic approach the central concept is that of *proof*: to show that a statement follows from some others one has to make use of a *deduction system*. Deduction systems are syntactic in nature, caring only about the form of sentences and not about what they represent or their possible meaning. One such system, the *axiomatic method*, distinguishes a subset of formulas, called *axioms*, formed by all sentences that match any of the following patterns:

$$\begin{array}{l}\varphi \to (\psi \to \varphi)\\ (\varphi \to (\psi \to \chi)) \to ((\varphi \to \psi) \to (\varphi \to \chi))\\ (\neg\varphi \to \neg\psi) \to (\psi \to \varphi)\end{array}$$

Given a set of formulas Γ, a formula φ is said to be a *logical consequence* of Γ, written $\Gamma \vdash \varphi$, if there is a sequence of formulas $\varphi_1, \varphi_2, \ldots, \varphi_n$ such that:

1. $\varphi_n = \varphi$.
2. For all $i \leq n$, either φ_i is an axiom, or φ_i belongs to Γ, or there exist $j, k < i$ such that $\varphi_k = \varphi_j \to \varphi_i$.

This corresponds to an iterative process in which we can add to the set of provable sentences, at every stage, either an axiom (whose validity is clear according to the defined semantics), an element of Γ (a hypothesis we are assuming as given), or a formula φ_i whenever we have previously proved φ_j and that φ_j implies φ_i.

The axiomatic method is not the only deduction system. In *natural deduction* there are rules associated to the connectives: introduction rules, to prove formulas containing the connective, and elimination rules, to obtain consequences from a formula with a given connective. For example, the following are prototypical:

$$\frac{\varphi \quad \psi}{\varphi \wedge \psi} \qquad \frac{\varphi \wedge \psi}{\varphi} \qquad \frac{\varphi \wedge \psi}{\psi} \qquad \frac{}{\varphi} \text{ if } \varphi \in \Gamma$$

The first rule is the introduction rule for the logical "and," and captures the idea that if φ and ψ can both be proved, so can their conjunction; the next two are elimination rules, stating that from a conjunction both its conjuncts can be derived; the last allows the derivation of a hypothesis. Similar rules are associated to the remaining connectives. Compared to the axiomatic method, natural deduction has a richer set of rules: as a result, it is easier to prove things *in* the logic using natural deduction, but the simplicity

of the axiomatic method comes in handy if one is interested in proving something *about* the logic. Although the presentations of deduction systems vary, they all allow the derivation of the same set of sentences (assuming they are properly designed).

Let us return to the example of the chef and the cake. In symbols, the argument can now be expressed as

$$\{(p \wedge \neg q) \to r,\, p\} \models \neg r \to q\,.$$

Although it is a bit tedious, one can consider all eight possible assignments of values to p, q, and r and check that it is actually a semantic consequence.

But then the following question can be raised. Why cannot the argument be expressed instead as

$$\{(p \wedge \neg q) \to r,\, p\} \vdash \neg r \to q?$$

Indeed, we have defined two different notions of consequence, semantic and logical, and though both are reasonable they do not seem to have much in common: which one should be chosen? This is an important *metalogical* question. Fortunately, in the case of propositional logic it does not matter for it can be proved that

$$\Gamma \vdash \varphi \quad \text{if and only if} \quad \Gamma \models \varphi\,.$$

The implication from left to right, that asserts that any proof-theoretic logical consequence is also a semantic consequence, is known as the *soundness of propositional logic*. The implication from right to left, that claims that any semantic consequence has a syntactic derivation, is the *completeness of propositional logic*.

Assume that we have a finite set Γ of assumptions. We then have two methods at our disposal to decide whether a given formula follows from Γ: either we build a syntactic proof, or show that all models of the assumptions also satisfy the formula. In particular, since we are working with a finite set of formulas there is only a finite number of atomic propositions involved: we can consider all possible valuations and study whether there is one that satifies Γ but not φ. Hence, for propositional logic, the validity problem is *decidable*, in the precise sense that there is an effective procedure or algorithm that solves it (see Computability).

This ends the presentation of propositional logic. Summing up, the most important elements introduced, common to all logics are: a syntax which defines the language; a semantics to assign meaning to the formulas; logical and semantic consequences and relationships between them; and the validity problem.

3 Predicate logic

The simplicity of propositional logic comes at a price: its expressive power is rather limited; in particular, it cannot deal with syllogisms like that at the beginning of this article:

> There are horses in Spain.
> All horses are mammals.

implies that

> There are mammals in Spain.

In propositional logic we would have to formalize the first premise by means of an atomic proposition p, the second with q, and the conclusion, r, would not be a valid consequence.

To remedy this, predicate logic (also known as first-order logic) introduces *predicates* and *quantifiers*. Assume that we use $H(x)$, $S(x)$, and $M(x)$ to express that x is a horse, x dwells in Spain, and x is a mammal, respectively. Then, the syllogism can be presented as a valid argument in predicate logic as

$$\frac{\begin{array}{l}\exists x\,(H(x) \wedge S(x)) \\ \forall x\,(H(x) \to M(x))\end{array}}{\exists x\,(M(x) \wedge S(x))}$$

where the quantifier $\forall$ means "for all" and $\exists$ means "there exists."

But predicate logic goes beyond syllogisms. It not only allows multiple premises, but also predicates with an arbitrary number of arguments. For example, a statement like "the ancestor of an ancestor is also an ancestor" has no room in the syllogistic theory, whereas it can be dealt with in predicate logic by using a binary predicate $A(x, y)$ with the meaning x is an ancestor of y:

$$A(x, y) \wedge A(y, z) \to A(x, z)\,.$$

In predicate logic one can distinguish two levels: terms and formulas; terms denote individuals while formulas are statements about those individuals. Terms are constructed from a set of variables $(x_0, x_1, x_2, \ldots)$ and a set of constants and function symbols with arbitrary arities:

$$t ::= x \mid c \mid f(t_1, \ldots, t_2) \quad f \text{ function symbol of arity } n\,.$$

Thus, if *mother* is a unary function symbol, the fact that one's mother is an ancestor can be expressed with $A(\mathit{mother}(x), x)$.

Formulas, in turn, are constructed from terms and a set of predicate symbols:

$$\varphi ::= t_1 = t_2 \mid R(t_1, \ldots, t_n) \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \varphi \to \varphi \mid \varphi \leftrightarrow \varphi \mid \forall x\varphi \mid \exists x\varphi$$

where R is a predicate symbol of arity n. The resulting set of formulas depends on the concrete sets of function and predicate symbols, F and P: we will write $L(F, P)$ to denote the set of formulas, or language, built using the sets F and P, or just L if no ambiguity arises. For example, if *peter* is a constant that represents a concrete boy and S is a binary predicate that stands for "sibling," the sentence

$$\forall x\,(S(x, \mathit{peter}) \to \forall z\,(A(z, \mathit{peter}) \leftrightarrow A(z, x)))$$

expresses that *peter* has the same ancestors as his siblings.

For another, less contrived example, consider the function symbols 0 (constant), *suc* (unary), and $+$, $*$ (binary), and the predicate $<$ (binary), which give rise to the language of arithmetic for obvious reasons. Commutativity of addition is then expressed as

$$\forall x \forall y\, +(x, y) = +(y, x)\,.$$

Though awkward, this is the correct notation; however, we will usually stick to the standard infix notation and write $+(x, y)$ as $x + y$.

Note that quantifiers are variable binding operators in the same sense as the summation symbol $\sum$ in an expression like $\sum_{x=1}^{9} x$, where the variable x cannot "vary" and take any arbitrary value. (This will become clearer once the semantics of quantifiers is presented in the next section.) In $\forall x\, \varphi$ and $\exists x\, \varphi$, the formula φ is said to be the scope of the quantifier. Then, an occurrence of a variable in an arbitrary formula is said to be *free* if it falls under the scope of no quantifier; otherwise, that occurrence is called *bound*.

3.1 Semantics

First of all, the universe of discourse, the elements to be referred to by terms, has to be fixed; then, function and predicate symbols from the sets F and P have to be interpreted over it. More precisely, a structure $\mathcal{A}$ is a tuple

$$\langle A, c^A, \ldots, f^A, \ldots, R^A \ldots \rangle$$

such that

- A is a nonempty set;
- for every constant $c \in F$, $c^A \in A$;
- for every n-ary function symbol $f \in F$, f^A is a function from A^n to A;
- for every n-ary predicate symbol $R \in P$, R^A is a subset of A^n.

An obvious structure $\mathcal{N}$ for the language of arithmetic consists of the set $\mathbb{N}$ of natural numbers as universe, with 0^N the number zero, suc^N the successor operation, $+^N$ and $*^N$ addition and multiplication of natural numbers, and $<^N$ the "less-than" relation. Note, however, that the structure can be arbitrary. Another valid structure is $\mathcal{M}$, where:

- the universe M is the set of digits $0, 1, 2, \ldots, 9$;
- 0^M is the digit 9;
- suc^M returns 0 when applied to 9 and the following digit in the usual order otherwise;
- $+^M$, when applied to two digits, returns the smallest one;
- $*^M$ returns the greatest digit;
- $<^M$ is the set of all pairs (u, v) with u greater than v.

More generally, the set A can consist of letters, derivable functions, matrices, or whatever elements one chooses.

Before meaning can be ascribed to terms, yet another component is needed: an *assignment* mapping variables to elements of A. Then, an *interpretation* $\mathcal{I} = (\mathcal{A}, V)$ is

a pair formed by a structure and an assignment V. The meaning of a term in a given interpretation, that is, the individual it refers to in the universe, can now be defined recursively:

$$\mathcal{I}(x) = V(x)$$
$$\mathcal{I}(c) = c^A$$
$$\mathcal{I}(f(t_1, \ldots, t_n)) = f^A(\mathcal{I}(t_1), \ldots, \mathcal{I}(t_n))$$

Let us consider the previously defined structure $\mathcal{N}$ and let V be an assignment such that $V(x) = 3$ and $V(y) = 5$. In the interpretation $\mathcal{I} = (\mathcal{N}, V)$, $\mathcal{I}(x * suc(suc(0))) = 6$ and $\mathcal{I}(x + suc(y)) = 9$. On the other hand, in the interpretation $\mathcal{J} = (\mathcal{M}, W)$, where $W(x) = 3$ and $W(y) = 5$, those same two terms get very different meanings: $\mathcal{J}(x * suc(suc(0))) = 3$ and $\mathcal{J}(x + suc(y)) = 3$.

A last piece of machinery is needed. Given an assignment V, a variable x, and an element a of the universe, we write $V[a/x]$ for the assignment that maps x to a and coincides with V in the remaining variables. The truth value of a formula, 0 (false) or 1 (true), with respect to an interpretation $\mathcal{I}$ can finally be defined:

- $\mathcal{I}(t_1 = t_2) = 1$ if $\mathcal{I}(t_1) = \mathcal{I}(t_2)$, and 0 otherwise;
- $\mathcal{I}(R(t_1, \ldots, t_n)) = 1$ if $(t_1^A, \ldots, t_n^A) \in R^A$;
- $\mathcal{I}(\neg\varphi)$, $\mathcal{I}(\varphi \wedge \psi)$, $\mathcal{I}(\varphi \vee \psi)$, $\mathcal{I}(\varphi \to \psi)$, $\mathcal{I}(\varphi \leftrightarrow \psi)$ are defined analogously to the propositional case.
- $\mathcal{I}(\forall x\, \varphi) = 1$ if $\mathcal{J}(\varphi) = 1$ for all $a \in A$, where $\mathcal{J} = (A, V[a/x])$;
- $\mathcal{I}(\exists x\, \varphi) = 1$ if there exists $a \in A$ with $\mathcal{J}(\varphi) = 1$, where $\mathcal{J} = (A, V[a/x])$.

As in propositional logic, if $\mathcal{I}(\varphi)$ is 1 we say that $\mathcal{I}$ is a model of φ or that $\mathcal{I}$ satisfies φ, and denote it by $\mathcal{I} \models \varphi$. We write $\Gamma \models \varphi$ if every model of all formulas in Γ is also a model of φ.

Note that assignments are only needed to ascribe a meaning to free occurrences of variables: if there are none, the interpretation of a formula is the same regardless of the assignment. Now it can be checked that the formulas

$$\forall x\, (0 < suc(x)) \quad \text{and} \quad \forall x \exists y\, (x < y)$$

are both true in the interpretation $\mathcal{N}$, but false in $\mathcal{M}$.

3.2 Proof theory

Deduction systems for predicate logic extend those of propositional logic to take care of predicates and quantifiers. In the axiomatic method, the main difference arises from the extension of the set of axioms.

A valuation in predicate logic is a function from the set of formulas to the set $\{0, 1\}$ that respects the meaning of propositional connectives, that is, $f(\neg\varphi) = 1 - f(\varphi)$, $f(\varphi \wedge \psi) = 1$ if and only if $f(\varphi) = f(\psi) = 1$, ... The set of axioms is then the set of formulas with one of the following forms:

1. Formulas which are mapped to 1 by all valuations.

2. $\forall x\,(\varphi \to \psi) \to (\forall x\,\varphi \to \forall x\,\psi)$.

3. $\varphi \to \forall x\,\varphi$, with no free occurrences of x in φ.

4. $\exists x\,(x = t)$ where t is a term which does not contain x.

5. $t_1 = t_2 \to (\varphi \to \psi)$ where φ contains no quantifiers and ψ is obtained from φ by replacing an occurrence of t_1 in φ with t_2.

A derivation of φ from Γ is a sequence $\varphi_1, \ldots, \varphi_n$ such that:

1. $\varphi_n = \varphi$.

2. For all $i \leq n$, either:

   (a) φ_i is an axiom;

   (b) φ_i belongs to Γ;

   (c) there exist $j, k < i$ such that $\varphi_k = \varphi_j \to \varphi_i$;

   (d) there exists $j < i$ and a variable x such that φ_i is $\forall x\,\varphi_j$.

The formula φ is then a logical consequence of Γ and it is denoted with $\Gamma \vdash \varphi$.

Likewise, a system of natural deduction for predicate logic is obtained, essentially, by extending the propositional one with the rules

$$\frac{\varphi}{\forall x\,\varphi} \qquad \frac{\forall x\,\varphi}{\varphi[t/x]}$$

subject to a couple of technical conditions that are omitted and where $\varphi[t/x]$ means that every free occurrence of x in φ is replaced by t.

3.3 Completeness and decidability

Deduction systems are designed so that they are sound, that is, so that a logical consequence is also a semantic consequence, and it is not hard to show that this is the case in predicate logic. The converse implication, completeness, is much harder to prove but it was also shown to hold for predicate logic by Gödel [3] in 1929. Therefore, for a set of formulas Γ and a formula φ,

$$\Gamma \vdash \varphi \quad \text{if and only if} \quad \Gamma \models \varphi\,.$$

Faced with the problem of deciding whether a given formula is a consequence of a set of premises, we again have the same two alternatives as in propositional logic: either build a derivation or consider the models involved. Now, however, there is a crucial difference: the set of models is not finite and thus, in general, it will not be possible their study to conclude whether a formula is a semantic consequence of the premises. Hence, if we intend to obtain a mechanical procedure, an algorithm, to decide the validity of a formula we are only left with the proof-theoretic approach.

Assume that we want to determine whether a formula φ is valid, that is, whether $\vdash \varphi$ (it can be derived from no hypothesis). By using the axiomatic method we can enumerate all valid formulas. First, all derivations φ_1 which use axioms of length up

to, say 10, are listed; since there can only be a finite number of these, the process ends. Next, derivations φ_1 and φ_1, φ_2 with axioms of length up to 11 are considered; again, there is only a finite number of such derivations. In the next step, derivations of up to three steps with axioms of length less than or equal to 12 are listed; and so on. The process is tedious, but it is mechanizable and considers all posible derivations. If φ is valid, then it has a corresponding derivation and this procedure will eventually produce it. But what if φ is not valid? Then the procedure will not terminate and will offer no clue as to the validity of φ. Indeed, that it is no accident but an unavoidable shortcoming of any procedure to decide the validity of a formula is the content of the *undecidability theorem* proved independently by Church and Turing [4, 5] in 1936.

Note that this does not mean that it is not possible to decide whether a given formula is valid or not, but that it is not possible to develop a general procedure that always works. For example, if all predicates considered are monadic (take one argument) the resulting language is decidable and in this case a computer could be programmed to solve the validity problem.

4 A glimpse of other logics

4.1 Second-order and higher-order logic

In predicate logic, variables range over the elements of the universe in the structure; this is why it is also called first-order logic. But in mathematics it is often necessary to refer, not to single individuals, but to collections of these. As a result, it is sometimes convenient to consider an extension of predicate logic with second-order variables that range over subsets or, more generally, over n-ary relations of the universe.

The syntax and semantics of second-order logic are defined similarly to those of precicate logic. Now, if X is an n-ary variable and $t_1, \ldots, t_n$ are terms, $X(t_1, \ldots, t_n)$ is also a formula. Second-order logic is more expressive than predicate logic. For example, the structure of the natural numbers cannot be characterized by means of predicate formulas because the induction principle can only be approximated by means of all formulas of the form

$$\varphi(0) \wedge \forall x\, (\varphi(x) \to \varphi(suc(x))) \to \forall x\, \varphi\,.$$

In second-order logic, however, the induction principle is formally captured by the single formula

$$\forall X\, (X(0) \wedge \forall x\, (X(x) \to X(suc(x))) \to \forall x\, X(x))\,,$$

where X is a unary variable, and the structure of natural numbers is characterizable.

Second-order logic allows the expression of mathematical facts in a more natural way; however, this additional expressive power makes the logic much more complex, with many useful properties of predicate logic no longer holding. In particular, there is no deduction system both sound and complete; of course, this is no obstacle for setting up correct and useful (though incomplete) systems. Also, the validity problem is even more undecidable (in a precise technical sense, see Computability) than in the first-order case.

While second-order logic allows to quantify over predicates, higher-order logic (historically, proposed a couple of decades earlier than predicate logic) goes a step beyond

and considers, and allows quantification over, predicates that take other predicates as arguments, and predicates which take predicates that take predicates, ... The resulting logic is, again, very complex but extremely expressive and it has proved to be very useful in computer science.

4.2 Intuitionistic logic

In traditional (also called classical) mathematics, nonconstructive arguments are valid proof methods: it is possible to show that there has to exist an element satisfying a certain property without actually producing a witness to it, and the statement that either a proposition or its negation holds is admitted as true even if none of them has been proved. Some mathematicians object against such principles when working with infinite sets and advocate the practice of constructive procedures. In particular:

- a statement of the form $\exists x\, \varphi$ is not proved until a concrete term t has been constructed such that $\varphi[t/x]$ is proved;
- a proof of a disjunction $\varphi \vee \psi$ is a pair $\langle a, b\rangle$ such that if $a = 0$ then b proves φ, and if $a \neq 0$ then b proves ψ.

Intuitionistic logic captures the valid principles of inference for constructive mathematics. In this new setting, familiar statements cease to hold: for example, $\varphi \vee \neg\varphi$ is not universally true and, while the implication $(\neg\varphi \vee \neg\psi) \to \neg(\varphi \wedge \psi)$ can still be proved, it is not possible to strenghten it to a biconditional. Perhaps surprisingly, a deductive system for intuitionistic logic can be obtained from a classical one just by preventing the *law of double negation*

$$\neg\neg\varphi \to \varphi\,,$$

or any other equivalent to it, from being used as an axiom.

In this sense, intuitionistic logic is a subset of classical logic. On the other hand, classical logic can also be embedded in intuitionistic logic: for every formula φ, a formula ψ can be constructed such that φ is derivable in the classical calculus if and only if ψ is derivable in the intuitionistic one.

Since the deduction system has been restricted, it is still sound with respect to the semantics for predicate logic but it is no longer complete. A number of semantics for intuitionistic logic have been defined, the simplest of which is probably the one introduced by Kripke. For propositional intuitionistic logic, a *Kripke structure* is a partially ordered set $\langle K, \leq\rangle$ together with an assignment V of atomic propositions to the elements of K such that, if $k \leq k'$, $V(k) \subseteq V(k')$. One can think of the elements in K as stages in time and then $V(k)$ would be the "basic facts" known at instant k. Satisfaction of a formula by a model, called *forcing* and representing the knowledge at a certain stage, is defined recursively:

- k forces p if $p \in V(k)$.
- k forces $\neg\varphi$ if for no $k' \geq k$ does k' force φ.
- k forces $\varphi \wedge \psi$ if k forces φ and k forces ψ.
- k forces $\varphi \vee \psi$ if k forces φ or k forces ψ.

- k forces $\varphi \to \psi$ if, for every $k' \geq k$, if k' forces φ then k' forces ψ.

The intuition for all clauses except the second and the fifth is clear. One knows $\varphi \to \psi$ at instant k, even if none of φ or ψ is yet known, if one knows that for all future instants one can establish ψ if φ can be established. As for the negation, $\neg\varphi$ is known when no evidence for φ can be found at a later stage.

A Kripke structure forces a formula if all its elements do so, and the intuitionistic calculus is sound and complete with respect to forcing. Kripke structures and forcing can be extended to predicate logic so that the corresponding deduction system also becomes sound and complete. Like their classical counterparts, intuitionistic propositional logic is decidable whereas intuitionistic predicate logic is not.

4.3 Predicate logic in perspective

In mathematics, predicate logic has been a great success, to the point of being often deemed as *the* logic: it is in principle sufficient for mathematics, has a sound and complete deduction system, and satisfies important semantic results. Indeed Lindstrom's theorems show that there can be no logical system with more expressive power than predicate logic and with the same good semantic properties.

By contrast, computer science (and other fields) has seen a cornucopia of logics suited for different purposes. To cite a few:

- *Modal logic.* It is a logic to reason about concepts such as possibility or necessity.
- *Temporal logic.* It is a brand of modal logic with operators to talk about the passage of time.
- *Fuzzy logic*, to deal with approximate, or vague concepts such as the distinction between "warm" and "hot".
- *Probabilistic logic*, in which the truth values of formulas are probabilities.
- *Nonmonotonic logic.* In this logic, an established piece of knowledge may have to be retracted if additional facts are later known.

Moreover, these logics come in different flavors, usually admitting propositional, first-order, higher-order, and intuitionistic presentations, as well as combinations of these and many ad hoc variants.

5 Logic and computer science

Though profound and important, the practical significance for mathematics of the results obtained during the blossoming of logic in the 20th century has been rather limited. By contrast, logic has risen to prominence in computer science where it is expected to play a role analogous to that of calculus in physics. Specification and programming rank among its most important areas of application, which also include fields as diverse as descriptive complexity, compiler techniques, databases, or type theory [6].

5.1 Specification and verification

Computer programs are extremely complex entities and reasoning about them, except for the smallest instances, is no easy feat. A given computer program poses two related problems: deciding what it is supposed to do and then checking whether it does it. Logic can help here, first, with the formal specification of the expected behavior of a program and, secondly, in the process of verifying that the program indeed abides by its specification.

In its first use, logic can be seen as tool to resolve ambiguities. Assume that a programmer is asked to write a program which, given two integers a and b, returns the quotient and remainder of dividing a by b. The behaviour of the program when a and b are positive should be obvious but if one of them is negative, say $a = 5$ and $b = -2$, and the mathematical training of the programmer is a bit shaky, he might be inclined to admit -3 and -1 as valid quotient and remainder. Furthermore, what should the behavior be when the divisor is 0? A bullet-proof, unambiguos specification of the behaviour of the program would look like:

$\varphi \equiv a, b$ integers
fun *division*(a, b) **returns** $\langle q, r \rangle$
$\psi \equiv (q, r \text{ integers}, a = b * q + r, 0 \leq r < |b|) \vee (b = 0 \wedge q = r = -1)$

In this specification, the *precondition* φ requires the arguments to be integers whereas the *postcondition* ψ imposes that q and r are the appropriate quotient and remainder if $b \neq 0$, and sets them both to -1 if b is 0 to mark the error condition. Alternatively, one could assume/require that b is never instantiated with 0 as value:

$\varphi \equiv a, b \text{ integers}, b \neq 0$
fun *division*(a, b) **returns** $\langle q, r \rangle$
$\psi \equiv q, r \text{ integers}, a = b * q + r, 0 \leq r < |b|$

In this case the postcondition ψ leaves unspecified the behaviour of the program if b is 0, so that it should be used only at one's own risk.

In general, the precondition imposes some requirements on the input while the postcondition states all properties that can be assumed about the output. Anything not reflected in them falls outside the programmer's responsibility. To express the precondition and postcondition, any logic can be used.

A specification imposes a contract on the programmer who, given that programming is an error-prone task, would like to have a means to verify that his final code actually satisfies those requirements (see Formal Program Verification). The prototypical example of the use of logic in this regard is *Hoare logic*, designed for imperative programming languages. It defines a derivation system consisting of a set of rules of the form

$$\frac{\text{Condition}}{\varphi \;\; \textit{instruction} \;\; \psi},$$

for every instruction in the programming language, establishing the conditions under which a precondition φ and postcondition ψ hold. For example, to the assignment instruction it corresponds the rule

$$\frac{}{\varphi[e/x] \;\; x := e \;\; \varphi}$$

which states that some property holds for variable x after the instruction is executed only if it already held when the expression e was substituted for x. Similarly, for sequential composition of instructions we have the rule

$$\frac{\varphi \;\; \mathit{intruction}_1 \;\; \psi \qquad \psi \;\; \mathit{intruction}_2 \;\; \chi}{\varphi \;\; \mathit{intruction}_1 \; ; \mathit{intruction}_2 \;\; \chi} \,,$$

which states the conditions required for χ to hold after executing $\mathit{intruction}_1$ followed by $\mathit{instruction}_2$. Ideally, to show that a program P satisfies a specification $\varphi \; P \; \psi$ one would start with ψ and the final line of P and proceed backwards by applying the corresponding rule in the calculus. The fact that programs are hundreds of thousands of lines long and that the application of some of the rules require human intervention makes this direct approach unfeasible in practice.

A different flavor in which logic supports the verification process comes in the form of *proof assistants* or *theorem provers* (see Automated Theorem Proving). These are usually embedded within verification systems which allow to formally specify functions and predicates, to state mathematical theorems and to develop formal proofs. Many of these systems are based on higher-order logic, but some use predicate logic. These environments do not directly work with the program code but instead focus on the algorithms that implements, by translating them into an appropriate logical language and then formally proving the properties they are required to satisfy. These systems are very powerful and some impressive results have been achieved in the verification of certain hardware architectures and protocols, but they present as a major drawback their dependency on user interaction.

In contrast to theorem provers, model checkers are fully automated and their underlying logic is some variation of temporal logic. *Model checking* was proposed in the early 1980s to verify complex hardware controllers and has also come to be used in the verification of software since then. A model checker explores all posible states in a system to check for a counterexample of the desired property and herein lies its limitation: whereas hardware controllers have a finite, if huge, number of states, typical programs have an infinite number. Even for hardware systems, the number of states can grow exponentially giving rise to what is known as the *explosion problem*. Hence, the devise of abstraction techniques that significantly reduce the size of a system to make it amenable to model checking, without altering its "main" properties, is an active area of research.

5.2 Programming

Logic also serves as the foundation of programming languages and has given rise to the *declarative paradigm*.

5.2.1 Logic programming

Imagine a problem for which all assumptions and requirements have been expressed as predicate logic formulas and gathered in a set Γ. For this problem, we are interested in determining whether there exists a solution, an element that under the given requirements satisfies a certain condition. Formally, we are interested in the entailment

$$\Gamma \vdash \exists x \, \varphi(x) \,.$$

Furthermore, we are probably interested not only in finding out if such an element exists but also in a concrete instance, that is, a term t such that $\Gamma \vdash \varphi[t/x]$.

In general, from $\Gamma \vdash \exists x\, \varphi$ it does not follow that $\varphi[t/x]$ for some term t; think of $\Gamma = \{\exists x\, R(x)\}$ for a counterexample. *Logic programming* is the area of research that studies the conditions that guarantee the existence of t and the ways of obtaining it (see Answer Set Programming). In logic programming, the formulas in Γ are restricted to *universal Horn formulas* of the forms

$$\forall x_1 \ldots \forall x_n\, \varphi \quad \text{and} \quad \forall x_1 \ldots \forall x_n\, (\varphi_1 \wedge \ldots \wedge \varphi_m \to \varphi)$$

while the goal is an existential formula

$$\exists x_1 \ldots \exists x_n\, (\varphi_1 \wedge \ldots \wedge \varphi_m)\,;$$

all φ_i and φ have the form $t = t'$ or $R(t_1, \ldots, t_n)$. Under these conditions, if the goal can be proved, then there exist concrete terms $t_1, \ldots, t_n$ for which φ holds. In principle, these terms can be found by systematically applying the rules of any of the deduction systems presented in Section 3. However, given the restricted form of Horn formulas, a rule of inference more suitable for logic programming called *resolution* has been developed which computes all terms that make $\varphi_1 \wedge \ldots \wedge \varphi_m$ true. Recall that the validity problem in predicate logic is undecidable; this is reflected in logic programming in the fact that every implementation of resolution may loop forever if there are no solutions to the problem.

As an example of a logic program, let us consider the problem of finding paths in a directed graph. Assuming we use constants $a, b, \ldots, f$ to represent the nodes and corresponding binary predicates *arc* and *path*, the conditions of the problem can be represented as follows (omitting quantifiers, as customary in this context):

$$\begin{array}{l} \mathit{arc}(a, b) \\ \mathit{arc}(a, c) \\ \mathit{arc}(b, d) \\ \mathit{arc}(e, f) \\ \mathit{arc}(f, e) \\ \mathit{arc}(x, y) \to \mathit{path}(x, y) \\ \mathit{arc}(x, z) \wedge \mathit{path}(z, y) \to \mathit{path}(x, y) \end{array}$$

A path from x to y is specified either as a direct arc between the nodes, or as an arc from x to an intermediate node z followed by a path from z to y. Now, to obtain all nodes reachable from a the goal $\mathit{path}(a, x)$ would be used and the resolution procedure would return b, c, and d as possible values for x.

Imposing further restrictions on the form of the formulas has led to the definition of query languages for deductive databases.

5.2.2 Functional programming

Functional programming languages are based on the *lambda calculus*. Although there are many variants and they all favour an operational interpretation, lambda calculi can be endowed with both semantics and sound and complete deduction systems (but note

that it took almost 40 years to define a semantics for the original, untyped lambda calculus), thus partaking in the main logical features. Unlike predicate logic, functions are first-class citizens in the lambda calculus and can therefore be passed as arguments to other functions and returned as results.

Functional languages come in many different flavors (see Functional Programming) but the central notion in all of them is that of definition $t \equiv s$, which is interpreted operationally as a rule that allows the transformation of the term in the lefthand side, t, into the one in the righthand side, s, in a process called *rewriting*. A program consists simply of a set of definitions. For example, the program

$$\begin{array}{ll} \mathit{square} : \mathit{Integer} \to \mathit{Integer} & \mathit{apply} : (\mathit{Integer} \to \mathit{Integer}) \to \mathit{Integer} \\ \mathit{square}\ x \equiv x * x & \mathit{apply}\ f\ x \equiv f\ x \end{array}$$

defines a function *square* that returns the square of its argument, and a function *apply* that takes a function f and an integer as arguments and applies f to that integer. The term *apply square* 2 would then be rewritten first to *square* 2 and then to $2 * 2$ (which the compiler would evaluate to 4). For some terms, however, the process of reduction may never stop:

$$\begin{array}{l} \mathit{infinity} : \mathit{Integer} \\ \mathit{infinity} \equiv \mathit{infinity} + 1 \end{array}$$

Terms such as this do not denote well-defined values in the normal mathematical sense. Another potential difficulty lies in the possible existence of different rewriting sequences for the same term. Given definitions $t \equiv t'$ and $t \equiv t''$, in principle t could be reduced to either t' or t'', and there are no guarantees that these two terms can be further reduced to a common one. In functional programming, if two different rewriting sequences terminate then they *converge*, that is, they lead to the same result.

5.2.3 Other paradigms and tidbits

Logic and functional programming are the two main representatives of declarative programming. Their main advantage is their logical foundation, which makes it possible to mathematically reason about the programs themselves and enormously facilitates the verification process. This has led many researchers to seek ways of enhancing their expressive power while remaining in the corresponding framework and thus retaining their good properties, and has produced languages in which convergence may not happen and nondeterminism is allowed or where (some) a priori infinite computations can be dealt with. Also, both approaches have been integrated in what is known as *functional logic programming*; here the computational process is guided not by resolution nor rewriting, but by a technique called *narrowing*.

Some languages have withdrawn from predicate logic and the lambda calculus and are based on *equational logic*, which is quite a restrictive subset of them both and precludes the use of functions. The loss in expressivity is made up with the gain in efficiency and simplicity of the mathematical reasoning about the programs. More recently, a whole family of languages has been designed based on yet another logic, *rewriting logic*, which extends equational logic with rules to allow a natural treatment of concurrency and nondeterminism. Unlike equations, a rule $t \to s$ does not express the identity between

the meanings of the two terms but rather that the state represented by t evolves to that represented by s.

6 Coda

The previous sections may convey the impression that a logic is a precisely defined entity, when there is actually no consensus about what constitutes a logic. It is true that most logics have both a model and a proof-theory, but that is not always the case and, even when it is, one of the approaches may be clearly emphasized over the other. For Quine [7], on more philosophical grounds, even a full-fledged logic like second-order logic should be rather regarded as a mathematical theory since its logical truths by themselves capture substantial mathematical statements. Together with the explosion in the number of proposed logics, this diversity has spurred work in research with a unifying aim that has resulted in the development of:

- logical frameworks, that are logics in which other logics can be represented and used, and
- extremely abstract formalisms such as *institutions*, that intend to capture the defining characteristics of any logic.

A detailed discussion on these topics, as well as on many others not mentioned in this article, can be found in the comprehensive surveys [8, 9, 10].

Much of the impetus for the development of mathematical logic came from the desire of providing a solid foundation for mathematics. Nowadays it is computer science that has taken the leading role and it will be the applications and needs in this area that are bound to guide the future of formal logic.

References

[1] M. Davis. *Engines of Logic: Mathematicians and the Origin of the Computer, 2nd edition*. W. W. Norton & Company, 2001.

[2] A. Tarski. The concept of truth in the languages of the deductive sciences (in Polish). *Prace Towarzystwa Naukowego Warszawskiego, Wydzial III Nauk Matematyczno-Fizycznych*, 34, 1933. Expanded English translation in [11].

[3] K. Gödel. The completeness of the axioms of the functional calculus of logic (in German). *Monatshefte für Mathematik und Physik*, 37:349–360, 1930. Reprinted in [12].

[4] A. Church. A note on the Entscheidungsproblem. *Journal of Symbolic Logic*, 1:40–41, 1936.

[5] A.M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936.

[6] J.Y. Halpern, R. Harper, N. Immerman, P.G. Kolaitis, M.Y. Vardi, and V. Vianu. On the unusual effectiveness of logic in computer science. *The Bulletin of Symbolic Logic*, 7(2):213–236, 2001.

[7] W.V. Quine. *Philosophy of Logic, 2nd edition*. Harvard University Press, 1986.

[8] S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors. *Handbook of Logic in Computer Science*. Oxford Science Publications. In six volumes, the first published in 1993.

[9] D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors. *Handbook of Logic in Artificial Intelligence and Logic Programming*. Oxford Science Publications. In five volumes, the first published in 1993.

[10] D.M. Gabbay and F. Guenthner, editors. *Handbook of Philosophical Logic. 2nd edition*. Springer. In eighteen volumes, the first published in 2001.

[11] A. Tarski. *Logic, Semantics, Metamathematics, papers from 1923 to 1938*. Hackett Publishing Company, 1983.

[12] K. Gödel. *Collected Works. I: Publications 1929–1936*. Oxford University Press, 1986.

Reading list

D. van Dalen. *Logic and Structure. Fourth Edition*. Springer, 2004.

H.-D. Ebbinghaus, J. Flum, and W. Thomas. *Mathematical Logic. Second Edition*. Springer, 1996.

J.A. Goguen and G. Malcolm. *Algebraic Semantics of Imperative Programs*. The MIT Press, 1996.