

Decision procedures in Algebra and Logic

Reading material

Contents

Articles

| | |
|---|-----------|
| Some background | 1 |
| Algebraic structure | 1 |
| Mathematical logic | 9 |
| Structure (mathematical logic) | 22 |
| Universal algebra | 29 |
| Model theory | 34 |
| Proof theory | 41 |
| Sequent calculus | 44 |
| Python (programming language) | 50 |
| Sage (mathematics software) | 62 |
| Decision procedures and implementations | 69 |
| Decision problem | 69 |
| Boolean satisfiability problem | 72 |
| Constraint satisfaction | 79 |
| Rewriting | 82 |
| Maude system | 87 |
| Resolution (logic) | 94 |
| Automated theorem proving | 98 |
| Prover9 | 103 |
| Method of analytic tableaux | 104 |
| Natural deduction | 124 |
| Isabelle (theorem prover) | 138 |
| Satisfiability Modulo Theories | 140 |
| Prolog | 144 |

References

| | |
|--|-----|
| Article Sources and Contributors | 157 |
| Image Sources, Licenses and Contributors | 159 |

Article Licenses

| | |
|---------|-----|
| License | 160 |
|---------|-----|

Some background

Algebraic structure

In algebra, a branch of pure mathematics, an **algebraic structure** consists of one or more sets closed under one or more operations, satisfying some axioms. Abstract algebra is primarily the study of algebraic structures and their properties. The notion of algebraic structure has been formalized in universal algebra.

As an abstraction, an "algebraic structure" is the collection of all possible models of a given set of axioms. More concretely, an algebraic structure is any particular model of some set of axioms. For example, the monster group both "is" an algebraic structure in the concrete sense, and abstractly, "has" the group structure in common with all other groups. This article employs both meanings of "structure."

This definition of an algebraic structure should not be taken as restrictive. Anything that satisfies the axioms defining a structure is an instance of that structure, regardless of how many other axioms that instance happens to have. For example, all groups are also semigroups and magmas.

Structures whose axioms are all identities

If the axioms defining a structure are all identities, the structure is a variety (not to be confused with algebraic variety in the sense of algebraic geometry). Identities are equations formulated using only the operations the structure allows, and variables that are tacitly universally quantified over the relevant universe. Identities contain no connectives, existentially quantified variables, or relations of any kind other than the allowed operations. The study of varieties is an important part of universal algebra.

All structures in this section are varieties. Some of these structures are most naturally axiomatized using one or more nonidentities, but are nevertheless varieties because there exists an equivalent axiomatization, one perhaps less perspicuous, composed solely of identities. Algebraic structures that are not varieties are described in the following section, and differ from varieties in their metamathematical properties.

In this section and the following one, structures are listed in approximate order of increasing complexity, operationalized as follows:

- *Simple* structures requiring but one set, the universe S , are listed before *composite* ones requiring two sets;
- Structures having the same number of required sets are then ordered by the number of binary operations (0 to 4) they require. Incidentally, no structure mentioned in this entry requires an operation whose arity exceeds 2;
- Let A and B be the two sets that make up a composite structure. Then a composite structure may include 1 or 2 functions of the form $A \times A \rightarrow B$ or $A \times B \rightarrow A$;
- Structures having the same number and kinds of binary operations and functions are more or less ordered by the number of required unary and 0-ary (distinguished elements) operations, 0 to 2 in both cases.

The indentation structure employed in this section and the one following is intended to convey information. If structure B is under structure A and more indented, then all theorems of A are theorems of B ; the converse does not hold.

Ringoids and lattices can be clearly distinguished despite both having two defining binary operations. In the case of ringoids, the two operations are linked by the distributive law; in the case of lattices, they are linked by the absorption law. Ringoids also tend to have numerical models, while lattices tend to have set-theoretic models.

Simple structures: No binary operation:

- Set: a degenerate algebraic structure having no operations.
-

- Pointed set: S has one or more distinguished elements, often 0, 1, or both.
- Unary system: S and a single unary operation over S .
- Pointed unary system: a unary system with S a pointed set.

Group-like structures:

One binary operation, denoted by concatenation. For monoids, boundary algebras, and sloops, S is a pointed set.

- Magma or groupoid: S and a single binary operation over S .
 - Steiner magma: A commutative magma satisfying $x(xy) = y$.
 - Squag: an idempotent Steiner magma.
 - Sloop: a Steiner magma with distinguished element 1, such that $xx = 1$.
- Semigroup: an associative magma.
 - Monoid: a unital semigroup.
 - Group: a monoid with a unary operation, inverse, giving rise to an inverse element.
 - Abelian group: a commutative group.
- Band: a semigroup of idempotents.
 - Semilattice: a commutative band. The binary operation can be called either meet or join.
 - Boundary algebra: a unital semilattice (equivalently, an idempotent commutative monoid) with a unary operation, complementation, denoted by enclosing its argument in parentheses, giving rise to an inverse element that is the complement of the identity element. The identity and inverse elements bound S . Also, $x(xy) = x(y)$ holds.

Three binary operations. Quasigroups are listed here, despite their having 3 binary operations, because they are (nonassociative) magmas. Quasigroups feature 3 binary operations only because establishing the quasigroup cancellation property by means of identities alone requires two binary operations in addition to the group operation.

- Quasigroup: a cancellative magma. Equivalently, $\forall x, y \in S, \exists ! a, b \in S$, such that $xa = y$ and $bx = y$.
 - Loop: a unital quasigroup with a unary operation, inverse.
 - Moufang loop: a loop in which a weakened form of associativity, $(zx)(yz) = z(xy)z$, holds.
 - Group: an associative loop.

Lattice: **Two** or more binary operations, including meet and join, connected by the absorption law. S is both a meet and join semilattice, and is a pointed set if and only if S is bounded. Lattices often have no unary operations. Every true statement has a dual, obtained by replacing every instance of meet with join, and vice versa.

- Bounded lattice: S has two distinguished elements, the greatest lower bound and the least upper bound. Dualizing requires replacing every instance of one bound by the other, and vice versa.
 - Complemented lattice: a lattice with a unary operation, complementation, denoted by postfix ', giving rise to an inverse element. That element and its complement bound the lattice.
- Modular lattice: a lattice in which the modular identity holds.
 - Distributive lattice: a lattice in which each of meet and join distributes over the other. Distributive lattices are modular, but the converse does not hold.
 - Kleene algebra: a bounded distributive lattice with a unary operation whose identities are $x'' = x$, $(x+y)' = x'y'$, and $(x+x')yy' = yy'$. See "ring-like structures" for another structure having the same name.
 - Boolean algebra: a complemented distributive lattice. Either of meet or join can be defined in terms of the other and complementation.
 - Interior algebra: a Boolean algebra with an added unary operation, the interior operator, denoted by postfix ' and obeying the identities $x'x = x$, $x'' = x$, $(xy)' = x'y'$, and $1' = 1$.

- Heyting algebra: a bounded distributive lattice with an added binary operation, relative pseudo-complement, denoted by infix " ' ", and governed by the axioms $x'x=1$, $x(x'y) = xy$, $x'(yz) = (x'y)(x'z)$, $(xy)'z = (x'z)(y'z)$.

Ringoids: Two binary operations, addition and multiplication, with multiplication distributing over addition. Semirings are pointed sets.

- Semiring: a ringoid such that S is a monoid under each operation. Each operation has a distinct identity element. Addition also commutes, and has an identity element that annihilates multiplication.
- Commutative semiring: a semiring with commutative multiplication.
- Ring: a semiring with a unary operation, additive inverse, giving rise to an inverse element equal to the additive identity element. Hence S is an abelian group under addition.
- Rng: a ring lacking a multiplicative identity.
- Commutative ring: a ring with commutative multiplication.
 - Boolean ring: a commutative ring with idempotent multiplication, equivalent to a Boolean algebra.
- Kleene algebra: a semiring with idempotent addition and a unary operation, the Kleene star, denoted by postfix $*$ and obeying the identities $(1+x^*x)x^*=x^*$ and $(1+xx^*)x^*=x^*$. See "Lattice-like structures" for another structure having the same name.

N.B. The above definition of ring does not command universal assent. Some authorities employ "ring" to denote what is here called a rng, and refer to a ring in the above sense as a "ring with identity."

Modules: Composite Systems Defined over Two Sets, M and R : The members of:

1. R are scalars, denoted by Greek letters. R is a ring under the binary operations of scalar addition and multiplication;
2. M are *module elements* (often but not necessarily vectors), denoted by Latin letters. M is an abelian group under addition. There may be other binary operations.

The *scalar multiplication* of scalars and module elements is a function $R \times M \rightarrow M$ which commutes, associates ($\forall r, s \in R, \forall x \in M, r(sx) = (rs)x$), has 1 as identity element, and distributes over module and scalar addition. If only the pre(post)multiplication of module elements by scalars is defined, the result is a *left (right) module*.

- Free module: a module having a free basis, $\{e_1, \dots, e_n\} \subset M$, where the positive integer n is the dimension of the free module. For every $v \in M$, there exist $\kappa_1, \dots, \kappa_n \in R$ such that $v = \kappa_1 e_1 + \dots + \kappa_n e_n$. Let $\mathbf{0}$ and 0 be the respective identity elements for module and scalar addition. If $r_1 e_1 + \dots + r_n e_n = \mathbf{0}$, then $r_1 = \dots = r_n = 0$.
- Algebra over a ring (also *R-algebra*): a (free) module where R is a commutative ring. There is a second binary operation over M , called multiplication and denoted by concatenation, which distributes over module addition and is bilinear: $\alpha(xy) = (\alpha x)y = x(\alpha y)$.
- Jordan ring: an algebra over a ring whose module multiplication commutes, does not associate, and respects the Jordan identity.

Vector spaces, closely related to modules, are defined in the next section.

Structures with some axioms that are not identities

The structures in this section are not varieties because they cannot be axiomatized with identities alone. Nearly all of the nonidentities below are one of two very elementary kinds:

1. The starting point for all structures in this section is a "nontrivial" ring, namely one such that $S \neq \{0\}$, 0 being the additive identity element. The nearest thing to an identity implying $S \neq \{0\}$ is the nonidentity $0 \neq 1$, which requires that the additive and multiplicative identities be distinct.
2. Nearly all structures described in this section include identities that hold for all members of S except 0. In order for an algebraic structure to be a variety, its operations must be defined for all members of S ; there can be no partial operations.

Structures whose axioms unavoidably include nonidentities are among the most important ones in mathematics, e.g., fields and vector spaces. Moreover, much of theoretical physics can be recast as models of multilinear algebras. Although structures with nonidentities retain an undoubted algebraic flavor, they suffer from defects varieties do not have. For example, neither the product of integral domains nor a free field over any set exist.

Arithmetics: **Two** binary operations, addition and multiplication. S is an infinite set. Arithmetics are pointed unary systems, whose unary operation is injective successor, and with distinguished element 0.

- Robinson arithmetic. Addition and multiplication are recursively defined by means of successor. 0 is the identity element for addition, and annihilates multiplication. Robinson arithmetic is listed here even though it is a variety, because of its closeness to Peano arithmetic.
- Peano arithmetic. Robinson arithmetic with an axiom schema of induction. Most ring and field axioms bearing on the properties of addition and multiplication are theorems of Peano arithmetic or of proper extensions thereof.

Field-like structures: **Two** binary operations, addition and multiplication. S is nontrivial, i.e., $S \neq \{0\}$.

- Domain: a ring whose sole zero divisor is 0.
 - Integral domain: a domain whose multiplication commutes. Also a commutative cancellative ring.
 - Euclidean domain: an integral domain with a function $f: S \rightarrow \mathbb{N}$ satisfying the division with remainder property.
- Division ring (or *sfield*, *skew field*): a ring in which every member of S other than 0 has a two-sided multiplicative inverse. The nonzero members of S form a group under multiplication.
- Field: a division ring whose multiplication commutes. The nonzero members of S form an abelian group under multiplication.
 - Ordered field: a field whose elements are totally ordered.
 - Real field: a Dedekind complete ordered field.

The following structures are not varieties for reasons in addition to $S \neq \{0\}$:

- Simple ring: a ring having no ideals other than 0 and S .
 - Weyl algebra:
- Artinian ring: a ring whose ideals satisfy the descending chain condition.

Composite Systems: Vector Spaces, and Algebras over Fields. Two Sets, M and R , and at least **three** binary operations.

The members of:

1. M are vectors, denoted by lower case letters. M is at minimum an abelian group under vector addition, with distinguished member $\mathbf{0}$.
2. R are scalars, denoted by Greek letters. R is a field, nearly always the real or complex field, with 0 and 1 as distinguished members.

Three binary operations.

- Vector space: a free module of dimension n except that R is a field.
- Normed vector space: a vector space with a norm, namely a function $M \rightarrow R$ that is positive homogeneous, subadditive, and positive definite.
 - Inner product space (also *Euclidean* vector space): a normed vector space such that R is the real field, whose norm is the square root of the inner product, $M \times M \rightarrow R$. Let i, j , and n be positive integers such that $1 \leq i, j \leq n$. Then M has an orthonormal basis such that $e_i \bullet e_j = 1$ if $i=j$ and 0 otherwise; see free module above.
 - Unitary space: Differs from inner product spaces in that R is the complex field, and the inner product has a different name, the hermitian inner product, with different properties: conjugate symmetric, bilinear, and

positive definite. See Birkhoff and MacLane (1979: 369).

- Graded vector space: a vector space such that the members of M have a direct sum decomposition. See graded algebra below.

Four binary operations.

- Algebra over a field: An algebra over a ring except that R is a field instead of a commutative ring.
- Jordan algebra: a Jordan ring except that R is a field.
- Lie algebra: an algebra over a field respecting the Jacobi identity, whose vector multiplication, the Lie bracket denoted $[u, v]$, anticommutes, does not associate, and is nilpotent.
- Associative algebra: an algebra over a field, or a module, whose vector multiplication associates.
- Linear algebra: an associative unital algebra with the members of M being matrices. Every matrix has a dimension $n \times m$, n and m positive integers. If one of n or m is 1, the matrix is a vector; if both are 1, it is a scalar. Addition of matrices is defined only if they have the same dimensions. Matrix multiplication, denoted by concatenation, is the vector multiplication. Let matrix A be $n \times m$ and matrix B be $i \times j$. Then AB is defined if and only if $m=i$; BA , if and only if $j=n$. There also exists an $m \times m$ matrix I and an $n \times n$ matrix J such that $AI=JA=A$. If u and v are vectors having the same dimensions, they have an inner product, denoted $\langle u, v \rangle$. Hence there is an orthonormal basis; see inner product space above. There is a unary function, the determinant, from square ($n \times n$ for any n) matrices to R .
- Commutative algebra: an associative algebra whose vector multiplication commutes.
- Symmetric algebra: a commutative algebra with unital vector multiplication.

Composite Systems: Multilinear algebras. Two sets, V and K . **Four** binary operations:

1. The members of V are multivectors (including vectors), denoted by lower case Latin letters. V is an abelian group under multivector addition, and a monoid under outer product. The outer product goes under various names, and is multilinear in principle but usually bilinear. The outer product defines the multivectors recursively starting from the vectors. Thus the members of V have a "degree" (see graded algebra below). Multivectors may have an inner product as well, denoted $u \bullet v: V \times V \rightarrow K$, that is symmetric, linear, and positive definite; see inner product space above.
 2. The properties and notation of K are the same as those of R above, except that K may have -1 as a distinguished member. K is usually the real field, as multilinear algebras are designed to describe physical phenomena without complex numbers.
 3. The multiplication of scalars and multivectors, $V \times K \rightarrow V$, has the same properties as the multiplication of scalars and module elements that is part of a module.
- Graded algebra: an associative algebra with unital outer product. The members of V have a direct sum decomposition resulting in their having a "degree," with vectors having degree 1. If u and v have degree i and j , respectively, the outer product of u and v is of degree $i+j$. V also has a distinguished member 0 for each possible degree. Hence all members of V having the same degree form an abelian group under addition.
 - Exterior algebra (also *Grassmann algebra*): a graded algebra whose anticommutative outer product, denoted by infix \wedge , is called the exterior product. V has an orthonormal basis. $v_1 \wedge v_2 \wedge \dots \wedge v_k = 0$ if and only if v_1, \dots, v_k are linearly dependent. Multivectors also have an inner product.
 - Clifford algebra: an exterior algebra with a symmetric bilinear form $Q: V \times V \rightarrow K$. The special case $Q=0$ yields an exterior algebra. The exterior product is written $\langle u, v \rangle$. Usually, $\langle e_i, e_i \rangle = -1$ (usually) or 1 (otherwise).
 - Geometric algebra: an exterior algebra whose exterior (called *geometric*) product is denoted by concatenation. The geometric product of parallel multivectors commutes, that of orthogonal vectors anticommutes. The product of a scalar with a multivector commutes. vv yields a scalar.
 - Grassmann-Cayley algebra: a geometric algebra without an inner product.

Examples

Some recurring universes: **N**=natural numbers; **Z**=integers; **Q**=rational numbers; **R**=real numbers; **C**=complex numbers.

N is a pointed unary system, and under addition and multiplication, is both the standard interpretation of Peano arithmetic and a commutative semiring.

Boolean algebras are at once semigroups, lattices, and rings. They would even be abelian groups if the identity and inverse elements were identical instead of complements.

Group-like structures

- Nonzero **N** under addition (+) is a magma.
- **N** under addition is a magma with an identity.
- **Z** under subtraction (−) is a quasigroup.
- Nonzero **Q** under division (÷) is a quasigroup.
- Every group is a loop, because $a * x = b$ if and only if $x = a^{-1} * b$, and $y * a = b$ if and only if $y = b * a^{-1}$.
- 2x2 matrices(of non-zero determinant) with matrix multiplication form a group.
- **Z** under addition (+) is an abelian group.
- Nonzero **Q** under multiplication (×) is an abelian group.
- Every cyclic group G is abelian, because if x, y are in G , then $xy = a^m a^n = a^{m+n} = a^{n+m} = a^n a^m = yx$. In particular, **Z** is an abelian group under addition, as is the integers modulo n $\mathbf{Z}/n\mathbf{Z}$.
- A monoid is a category with a single object, in which case the composition of morphisms and the identity morphism interpret monoid multiplication and identity element, respectively.
- The Boolean algebra **2** is a boundary algebra.
- More examples of groups and list of small groups.

Lattices

- The normal subgroups of a group, and the submodules of a module, are modular lattices.
- Any field of sets, and the connectives of first-order logic, are models of Boolean algebra.
- The connectives of intuitionistic logic form a model of Heyting algebra.
- The modal logic S4 is a model of interior algebra.
- Peano arithmetic and most axiomatic set theories, including ZFC, NBG, and New foundations, can be recast as models of relation algebra.

Ring-like structures

- The set $R[X]$ of all polynomials over some coefficient ring R is a ring.
- 2x2 matrices with matrix addition and multiplication form a ring.
- If n is a positive integer, then the set $\mathbf{Z}_n = \mathbf{Z}/n\mathbf{Z}$ of integers modulo n (the additive cyclic group of order n) forms a ring having n elements (see modular arithmetic).

Integral domains

- **Z** under addition and multiplication is an integral domain.
- The p-adic integers.

Fields

- Each of **Q**, **R**, and **C**, under addition and multiplication, is a field.
- **R** totally ordered by "<" in the usual way is an ordered field and is categorical. The resulting real field grounds real and functional analysis.
 - **R** contains several interesting subfields, the algebraic, the computable, and the definable numbers.
- An algebraic number field is a finite field extension of **Q**, that is, a field containing **Q** which has finite dimension as a vector space over **Q**. Algebraic number fields are very important in number theory.

- If $q > 1$ is a power of a prime number, then there exists (up to isomorphism) exactly one finite field with q elements, usually denoted \mathbf{F}_q , or in the case that q is itself prime, by $\mathbf{Z}/q\mathbf{Z}$. Such fields are called Galois fields, whence the alternative notation $\text{GF}(q)$. All finite fields are isomorphic to some Galois field.
- Given some prime number p , the set $\mathbf{Z}_p = \mathbf{Z}/p\mathbf{Z}$ of integers modulo p is the finite field with p elements: $\mathbf{F}_p = \{0, 1, \dots, p-1\}$ where the operations are defined by performing the operation in \mathbf{Z} , dividing by p and taking the remainder; see modular arithmetic.

Allowing additional structure

Algebraic structures can also be defined on sets with added structure of a non-algebraic nature, such as a topology. The added structure must be compatible, in some sense, with the algebraic structure.

- Ordered group: a group with a compatible partial order. I.e., S is partially ordered.
- Linearly ordered group: a group whose S is a linear order.
- Archimedean group: a linearly ordered group for which the Archimedean property holds.
- Lie group: a group whose S has a compatible smooth manifold structure.
- Topological group: a group whose S has a compatible topology.
- Topological vector space: a vector space whose M has a compatible topology; a superset of normed vector spaces.
- Banach spaces, Hilbert spaces, Inner product spaces
- Vertex operator algebras

Category theory

The discussion above has been cast in terms of elementary abstract and universal algebra. Category theory is another way of reasoning about algebraic structures (see, for example, Mac Lane 1998). A category is a collection of *objects* with associated *morphisms*. Every algebraic structure has its own notion of homomorphism, namely any function compatible with the operation(s) defining the structure. In this way, every algebraic structure gives rise to a category. For example, the category of groups has all groups as objects and all group homomorphisms as morphisms. This concrete category may be seen as a category of sets with added category-theoretic structure. Likewise, the category of topological groups (whose morphisms are the continuous group homomorphisms) is a category of topological spaces with extra structure.

There are various concepts in category theory that try to capture the algebraic character of a context, for instance

- algebraic
- essentially algebraic
- presentable
- locally presentable
- monadic functors and categories
- universal property.

See also

- arity
- category theory
- free object
- list of algebraic structures
- list of first order theories
- signature
- variety

References

- MacLane, Saunders; Birkhoff, Garrett (1999), *Algebra* (2nd ed.), AMS Chelsea, ISBN 978-0-8218-1646-2
- Michel, Anthony N.; Herget, Charles J. (1993), *Applied Algebra and Functional Analysis*, New York: Dover Publications, ISBN 978-0-486-67598-5

A monograph available free online:

- Burris, Stanley N.; Sankappanavar, H. P. (1981), *A Course in Universal Algebra* ^[1], Berlin, New York: Springer-Verlag, ISBN 978-3-540-90578-3

Category theory:

- Mac Lane, Saunders (1998), *Categories for the Working Mathematician* (2nd ed.), Berlin, New York: Springer-Verlag, ISBN 978-0-387-98403-2
- Taylor, Paul (1999), *Practical foundations of mathematics*, Cambridge University Press, ISBN 978-0-521-63107-5

External links

- Jipsen's algebra structures. ^[2] Includes many structures not mentioned here.
- Mathworld ^[3] page on abstract algebra.
- Stanford Encyclopedia of Philosophy: Algebra ^[4] by Vaughan Pratt.

References

[1] <http://www.thoralf.uwaterloo.ca/htdocs/ualg.html>

[2] <http://math.chapman.edu/cgi-bin/structures>

[3] <http://mathworld.wolfram.com/topics/Algebra.html>

[4] <http://plato.stanford.edu/entries/algebra/>

Mathematical logic

Mathematical logic (formerly known as **symbolic logic**) is a subfield of mathematics with close connections to computer science and philosophical logic.^[1] The field includes both the mathematical study of logic and the applications of formal logic to other areas of mathematics. The unifying themes in mathematical logic include the study of the expressive power of formal systems and the deductive power of formal proof systems.

Mathematical logic is often divided into the fields of set theory, model theory, recursion theory, and proof theory. These areas share basic results on logic, particularly first-order logic, and definability. In computer science (particularly in the ACM Classification) mathematical logic is seen as encompassing additional topics that are not detailed in this article; see logic in computer science for those.

Since its inception, mathematical logic has contributed to, and has been motivated by, the study of foundations of mathematics. This study began in the late 19th century with the development of axiomatic frameworks for geometry, arithmetic, and analysis. In the early 20th century it was shaped by David Hilbert's program to prove the consistency of foundational theories. Results of Kurt Gödel, Gerhard Gentzen, and others provided partial resolution to the program, and clarified the issues involved in proving consistency. Work in set theory showed that almost all ordinary mathematics can be formalized in terms of sets, although there are some theorems that cannot be proven in common axiom systems for set theory. Contemporary work in the foundations of mathematics often focuses on establishing which parts of mathematics can be formalized in particular formal systems, rather than trying to find theories in which all of mathematics can be developed.

History

Mathematical logic emerged in the mid-19th century as a subfield of mathematics independent of the traditional study of logic (Ferreirós 2001, p. 443). Before this emergence, logic was studied with rhetoric, through the syllogism, and with philosophy. The first half of the 20th century saw an explosion of fundamental results, accompanied by vigorous debate over the foundations of mathematics.

Early history

Sophisticated theories of logic were developed in many cultures, including China, India, Greece and the Islamic world. In the 18th century, attempts to treat the operations of formal logic in a symbolic or algebraic way had been made by philosophical mathematicians including Leibniz and Lambert, but their labors remained isolated and little known.

19th century

In the middle of the nineteenth century, George Boole and then Augustus De Morgan presented systematic mathematical treatments of logic. Their work, building on work by algebraists such as George Peacock, extended the traditional Aristotelian doctrine of logic into a sufficient framework for the study of foundations of mathematics (Katz 1998, p. 686).

Charles Sanders Peirce built upon the work of Boole to develop a logical system for relations and quantifiers, which he published in several papers from 1870 to 1885. Gottlob Frege presented an independent development of logic with quantifiers in his *Begriffsschrift*, published in 1879. Frege's work remained obscure, however, until Bertrand Russell began to promote it near the turn of the century. The two-dimensional notation Frege developed was never widely adopted and is unused in contemporary texts.

From 1890 to 1905, Ernst Schröder published *Vorlesungen über die Algebra der Logik* in three volumes. This work summarized and extended the work of Boole, De Morgan, and Peirce, and was a comprehensive reference to symbolic logic as it was understood at the end of the 19th century.

Foundational theories

Some concerns that mathematics had not been built on a proper foundation led to the development of axiomatic systems for fundamental areas of mathematics such as arithmetic, analysis, and geometry.

In logic, the term *arithmetic* refers to the theory of the natural numbers. Giuseppe Peano (1888) published a set of axioms for arithmetic that came to bear his name (Peano axioms), using a variation of the logical system of Boole and Schröder but adding quantifiers. Peano was unaware of Frege's work at the time. Around the same time Richard Dedekind showed that the natural numbers are uniquely characterized by their induction properties. Dedekind (1888) proposed a different characterization, which lacked the formal logical character of Peano's axioms. Dedekind's work, however, proved theorems inaccessible in Peano's system, including the uniqueness of the set of natural numbers (up to isomorphism) and the recursive definitions of addition and multiplication from the successor function and mathematical induction.

In the mid-19th century, flaws in Euclid's axioms for geometry became known (Katz 1998, p. 774). In addition to the independence of the parallel postulate, established by Nikolai Lobachevsky in 1826 (Lobachevsky 1840), mathematicians discovered that certain theorems taken for granted by Euclid were not in fact provable from his axioms. Among these is the theorem that a line contains at least two points, or that circles of the same radius whose centers are separated by that radius must intersect. Hilbert (1899) developed a complete set of axioms for geometry, building on previous work by Pasch (1882). The success in axiomatizing geometry motivated Hilbert to seek complete axiomatizations of other areas of mathematics, such as the natural numbers and the real line. This would prove to be a major area of research in the first half of the 20th century.

The 19th century saw great advances in the theory of real analysis, including theories of convergence of functions and Fourier series. Mathematicians such as Karl Weierstrass began to construct functions that stretched intuition, such as nowhere-differentiable continuous functions. Previous conceptions of a function as a rule for computation, or a smooth graph, were no longer adequate. Weierstrass began to advocate the arithmetization of analysis, which sought to axiomatize analysis using properties of the natural numbers. The modern (ϵ, δ) -definition of limit and continuous functions was already developed by Bolzano in 1817 (Felscher 2000), but remained relatively unknown. Cauchy in 1821 defined continuity in terms of infinitesimals (see *Cours d'Analyse*, page 34). In 1858, Dedekind proposed a definition of the real numbers in terms of Dedekind cuts of rational numbers (Dedekind 1872), a definition still employed in contemporary texts.

Georg Cantor developed the fundamental concepts of infinite set theory. His early results developed the theory of cardinality and proved that the reals and the natural numbers have different cardinalities (Cantor 1874). Over the next twenty years, Cantor developed a theory of transfinite numbers in a series of publications. In 1891, he published a new proof of the uncountability of the real numbers that introduced the diagonal argument, and used this method to prove Cantor's theorem that no set can have the same cardinality as its powerset. Cantor believed that every set could be well-ordered, but was unable to produce a proof for this result, leaving it as an open problem in 1895 (Katz 1998, p. 807).

20th century

In the early decades of the 20th century, the main areas of study were set theory and formal logic. The discovery of paradoxes in informal set theory caused some to wonder whether mathematics itself is inconsistent, and to look for proofs of consistency.

In 1900, Hilbert posed a famous list of 23 problems for the next century. The first two of these were to resolve the continuum hypothesis and prove the consistency of elementary arithmetic, respectively; the tenth was to produce a method that could decide whether a multivariate polynomial equation over the integers has a solution. Subsequent work to resolve these problems shaped the direction of mathematical logic, as did the effort to resolve Hilbert's *Entscheidungsproblem*, posed in 1928. This problem asked for a procedure that would decide, given a formalized mathematical statement, whether the statement is true or false.

Set theory and paradoxes

Ernst Zermelo (1904) gave a proof that every set could be well-ordered, a result Georg Cantor had been unable to obtain. To achieve the proof, Zermelo introduced the axiom of choice, which drew heated debate and research among mathematicians and the pioneers of set theory. The immediate criticism of the method led Zermelo to publish a second exposition of his result, directly addressing criticisms of his proof (Zermelo 1908a). This paper led to the general acceptance of the axiom of choice in the mathematics community.

Skepticism about the axiom of choice was reinforced by recently discovered paradoxes in naive set theory. Cesare Burali-Forti (1897) was the first to state a paradox: the Burali-Forti paradox shows that the collection of all ordinal numbers cannot form a set. Very soon thereafter, Bertrand Russell discovered Russell's paradox in 1901, and Jules Richard (1905) discovered Richard's paradox.

Zermelo (1908b) provided the first set of axioms for set theory. These axioms, together with the additional axiom of replacement proposed by Abraham Fraenkel, are now called Zermelo–Fraenkel set theory (ZF). Zermelo's axioms incorporated the principle of limitation of size to avoid Russell's paradox.

In 1910, the first volume of *Principia Mathematica* by Russell and Alfred North Whitehead was published. This seminal work developed the theory of functions and cardinality in a completely formal framework of type theory, which Russell and Whitehead developed in an effort to avoid the paradoxes. *Principia Mathematica* is considered one of the most influential works of the 20th century, although the framework of type theory did not prove popular as a foundational theory for mathematics (Ferreirós 2001, p. 445).

Fraenkel (1922) proved that the axiom of choice cannot be proved from the remaining axioms of Zermelo's set theory with urelements. Later work by Paul Cohen (1966) showed that the addition of urelements is not needed, and the axiom of choice is unprovable in ZF. Cohen's proof developed the method of forcing, which is now an important tool for establishing independence results in set theory.

Symbolic logic

Leopold Löwenheim (1915) and Thoralf Skolem (1920) obtained the Löwenheim–Skolem theorem, which says that first-order logic cannot control the cardinalities of infinite structures. Skolem realized that this theorem would apply to first-order formalizations of set theory, and that it implies any such formalization has a countable model. This counterintuitive fact became known as Skolem's paradox.

In his doctoral thesis, Kurt Gödel (1929) proved the completeness theorem, which establishes a correspondence between syntax and semantics in first-order logic. Gödel used the completeness theorem to prove the compactness theorem, demonstrating the finitary nature of first-order logical consequence. These results helped establish first-order logic as the dominant logic used by mathematicians.

In 1931, Gödel published *On Formally Undecidable Propositions of Principia Mathematica and Related Systems*, which proved the incompleteness (in a different meaning of the word) of all sufficiently strong, effective first-order theories. This result, known as Gödel's incompleteness theorem, establishes severe limitations on axiomatic foundations for mathematics, striking a strong blow to Hilbert's program. It showed the impossibility of providing a consistency proof of arithmetic within any formal theory of arithmetic. Hilbert, however, did not acknowledge the importance of the incompleteness theorem for some time.

Gödel's theorem shows that a consistency proof of any sufficiently strong, effective axiom system cannot be obtained in the system itself, if the system is consistent, nor in any weaker system. This leaves open the possibility of consistency proofs that cannot be formalized within the system they consider. Gentzen (1936) proved the consistency of arithmetic using a finitistic system together with a principle of transfinite induction. Gentzen's result introduced the ideas of cut elimination and proof-theoretic ordinals, which became key tools in proof theory. Gödel (1958) gave a different consistency proof, which reduces the consistency of classical arithmetic to that of intuitionistic arithmetic in higher types.

Beginnings of the other branches

Alfred Tarski developed the basics of model theory.

Beginning in 1935, a group of prominent mathematicians collaborated under the pseudonym Nicolas Bourbaki to publish a series of encyclopedic mathematics texts. These texts, written in an austere and axiomatic style, emphasized rigorous presentation and set-theoretic foundations. Terminology coined by these texts, such as the words *bijection*, *injection*, and *surjection*, and the set-theoretic foundations the texts employed, were widely adopted throughout mathematics.

The study of computability came to be known as recursion theory, because early formalizations by Gödel and Kleene relied on recursive definitions of functions.^[2] When these definitions were shown equivalent to Turing's formalization involving Turing machines, it became clear that a new concept – the computable function – had been discovered, and that this definition was robust enough to admit numerous independent characterizations. In his work on the incompleteness theorems in 1931, Gödel lacked a rigorous concept of an effective formal system; he immediately realized that the new definitions of computability could be used for this purpose, allowing him to state the incompleteness theorems in generality that could only be implied in the original paper.

Numerous results in recursion theory were obtained in the 1940s by Stephen Cole Kleene and Emil Leon Post. Kleene (1943) introduced the concepts of relative computability, foreshadowed by Turing (1939), and the arithmetical hierarchy. Kleene later generalized recursion theory to higher-order functionals. Kleene and Kreisel studied formal versions of intuitionistic mathematics, particularly in the context of proof theory.

Subfields and scope

The *Handbook of Mathematical Logic* makes a rough division of contemporary mathematical logic into four areas:

1. set theory
2. model theory
3. recursion theory, and
4. proof theory and constructive mathematics (considered as parts of a single area).

Each area has a distinct focus, although many techniques and results are shared between multiple areas. The border lines between these fields, and the lines between mathematical logic and other fields of mathematics, are not always sharp. Gödel's incompleteness theorem marks not only a milestone in recursion theory and proof theory, but has also led to Löb's theorem in modal logic. The method of forcing is employed in set theory, model theory, and recursion theory, as well as in the study of intuitionistic mathematics.

The mathematical field of category theory uses many formal axiomatic methods, and includes the study of categorical logic, but category theory is not ordinarily considered a subfield of mathematical logic. Because of its applicability in diverse fields of mathematics, mathematicians including Saunders Mac Lane have proposed category theory as a foundational system for mathematics, independent of set theory. These foundations use toposes, which resemble generalized models of set theory that may employ classical or nonclassical logic.

Formal logical systems

At its core, mathematical logic deals with mathematical concepts expressed using formal logical systems. These systems, though they differ in many details, share the common property of considering only expressions in a fixed formal language, or signature. The system of first-order logic is the most widely studied today, because of its applicability to foundations of mathematics and because of its desirable proof-theoretic properties.^[3] Stronger classical logics such as second-order logic or infinitary logic are also studied, along with nonclassical logics such as intuitionistic logic.

First-order logic

First-order logic is a particular formal system of logic. Its syntax involves only finite expressions as well-formed formulas, while its semantics are characterized by the limitation of all quantifiers to a fixed domain of discourse.

Early results about formal logic established limitations of first-order logic. The Löwenheim–Skolem theorem (1919) showed that if a set of sentences in a countable first-order language has an infinite model then it has at least one model of each infinite cardinality. This shows that it is impossible for a set of first-order axioms to characterize the natural numbers, the real numbers, or any other infinite structure up to isomorphism. As the goal of early foundational studies was to produce axiomatic theories for all parts of mathematics, this limitation was particularly stark.

Gödel's completeness theorem (Gödel 1929) established the equivalence between semantic and syntactic definitions of logical consequence in first-order logic. It shows that if a particular sentence is true in every model that satisfies a particular set of axioms, then there must be a finite deduction of the sentence from the axioms. The compactness theorem first appeared as a lemma in Gödel's proof of the completeness theorem, and it took many years before logicians grasped its significance and began to apply it routinely. It says that a set of sentences has a model if and only if every finite subset has a model, or in other words that an inconsistent set of formulas must have a finite inconsistent subset. The completeness and compactness theorems allow for sophisticated analysis of logical consequence in first-order logic and the development of model theory, and they are a key reason for the prominence of first-order logic in mathematics.

Gödel's incompleteness theorems (Gödel 1931) establish additional limits on first-order axiomatizations. The **first incompleteness theorem** states that for any sufficiently strong, effectively given logical system there exists a statement which is true but not provable within that system. Here a logical system is effectively given if it is possible to decide, given any formula in the language of the system, whether the formula is an axiom. A logical system is sufficiently strong if it can express the Peano axioms. When applied to first-order logic, the first incompleteness theorem implies that any sufficiently strong, consistent, effective first-order theory has models that are not elementarily equivalent, a stronger limitation than the one established by the Löwenheim–Skolem theorem. The **second incompleteness theorem** states that no sufficiently strong, consistent, effective axiom system for arithmetic can prove its own consistency, which has been interpreted to show that Hilbert's program cannot be completed.

Other classical logics

Many logics besides first-order logic are studied. These include infinitary logics, which allow for formulas to provide an infinite amount of information, and higher-order logics, which include a portion of set theory directly in their semantics.

The most well studied infinitary logic is $L_{\omega_1, \omega}$. In this logic, quantifiers may only be nested to finite depths, as in first order logic, but formulas may have finite or countably infinite conjunctions and disjunctions within them. Thus, for example, it is possible to say that an object is a whole number using a formula of $L_{\omega_1, \omega}$ such as

$$(x = 0) \vee (x = 1) \vee (x = 2) \vee \dots$$

Higher-order logics allow for quantification not only of elements of the domain of discourse, but subsets of the domain of discourse, sets of such subsets, and other objects of higher type. The semantics are defined so that, rather than having a separate domain for each higher-type quantifier to range over, the quantifiers instead range over all objects of the appropriate type. The logics studied before the development of first-order logic, for example Frege's logic, had similar set-theoretic aspects. Although higher-order logics are more expressive, allowing complete axiomatizations of structures such as the natural numbers, they do not satisfy analogues of the completeness and compactness theorems from first-order logic, and are thus less amenable to proof-theoretic analysis.

Another type of logics are fixed-point logics that allow inductive definitions, like one writes for primitive recursive functions.

One can formally define an extension of first-order logic — a notion which encompasses all logics in this section because they behave like first-order logic in certain fundamental ways, but does not encompass all logics in general, e.g. it does not encompass intuitionistic, modal or fuzzy logic. Lindström's theorem implies that the only extension of first-order logic satisfying both the Compactness theorem and the Downward Löwenheim–Skolem theorem is first-order logic.

Nonclassical and modal logic

Modal logics include additional modal operators, such as an operator which states that a particular formula is not only true, but necessarily true. Although modal logic is not often used to axiomatize mathematics, it has been used to study the properties of first-order provability (Solovay 1976) and set-theoretic forcing (Hamkins and Löwe 2007).

Intuitionistic logic was developed by Heyting to study Brouwer's program of intuitionism, in which Brouwer himself avoided formalization. Intuitionistic logic specifically does not include the law of the excluded middle, which states that each sentence is either true or its negation is true. Kleene's work with the proof theory of intuitionistic logic showed that constructive information can be recovered from intuitionistic proofs. For example, any provably total function in intuitionistic arithmetic is computable; this is not true in classical theories of arithmetic such as Peano arithmetic.

Algebraic logic

Algebraic logic uses the methods of abstract algebra to study the semantics of formal logics. A fundamental example is the use of Boolean algebras to represent truth values in classical propositional logic, and the use of Heyting algebras to represent truth values in intuitionistic propositional logic. Stronger logics, such as first-order logic and higher-order logic, are studied using more complicated algebraic structures such as cylindric algebras.

Set theory

Set theory is the study of sets, which are abstract collections of objects. Many of the basic notions, such as ordinal and cardinal numbers, were developed informally by Cantor before formal axiomatizations of set theory were developed. The first such axiomatization, due to Zermelo (1908b), was extended slightly to become Zermelo–Fraenkel set theory (ZF), which is now the most widely used foundational theory for mathematics.

Other formalizations of set theory have been proposed, including von Neumann–Bernays–Gödel set theory (NBG), Morse–Kelley set theory (MK), and New Foundations (NF). Of these, ZF, NBG, and MK are similar in describing a cumulative hierarchy of sets. New Foundations takes a different approach; it allows objects such as the set of all sets at the cost of restrictions on its set-existence axioms. The system of Kripke–Platek set theory is closely related to generalized recursion theory.

Two famous statements in set theory are the axiom of choice and the continuum hypothesis. The axiom of choice, first stated by Zermelo (1904), was proved independent of ZF by Fraenkel (1922), but has come to be widely accepted by mathematicians. It states that given a collection of nonempty sets there is a single set C that contains exactly one element from each set in the collection. The set C is said to "choose" one element from each set in the collection. While the ability to make such a choice is considered obvious by some, since each set in the collection is nonempty, the lack of a general, concrete rule by which the choice can be made renders the axiom nonconstructive. Stefan Banach and Alfred Tarski (1924) showed that the axiom of choice can be used to decompose a solid ball into a finite number of pieces which can then be rearranged, with no scaling, to make two solid balls of the original size. This theorem, known as the Banach–Tarski paradox, is one of many counterintuitive results of the axiom of choice.

The continuum hypothesis, first proposed as a conjecture by Cantor, was listed by David Hilbert as one of his 23 problems in 1900. Gödel showed that the continuum hypothesis cannot be disproven from the axioms of Zermelo–Fraenkel set theory (with or without the axiom of choice), by developing the constructible universe of set

theory in which the continuum hypothesis must hold. In 1963, Paul Cohen showed that the continuum hypothesis cannot be proven from the axioms of Zermelo–Fraenkel set theory (Cohen 1966). This independence result did not completely settle Hilbert's question, however, as it is possible that new axioms for set theory could resolve the hypothesis. Recent work along these lines has been conducted by W. Hugh Woodin, although its importance is not yet clear (Woodin 2001).

Contemporary research in set theory includes the study of large cardinals and determinacy. Large cardinals are cardinal numbers with particular properties so strong that the existence of such cardinals cannot be proved in ZFC. The existence of the smallest large cardinal typically studied, an inaccessible cardinal, already implies the consistency of ZFC. Despite the fact that large cardinals have extremely high cardinality, their existence has many ramifications for the structure of the real line. *Determinacy* refers to the possible existence of winning strategies for certain two-player games (the games are said to be *determined*). The existence of these strategies implies structural properties of the real line and other Polish spaces.

Model theory

Model theory studies the models of various formal theories. Here a theory is a set of formulas in a particular formal logic and signature, while a model is a structure that gives a concrete interpretation of the theory. Model theory is closely related to universal algebra and algebraic geometry, although the methods of model theory focus more on logical considerations than those fields.

The set of all models of a particular theory is called an elementary class; classical model theory seeks to determine the properties of models in a particular elementary class, or determine whether certain classes of structures form elementary classes.

The method of quantifier elimination can be used to show that definable sets in particular theories cannot be too complicated. Tarski (1948) established quantifier elimination for real-closed fields, a result which also shows the theory of the field of real numbers is decidable. (He also noted that his methods were equally applicable to algebraically closed fields of arbitrary characteristic.) A modern subfield developing from this is concerned with o-minimal structures.

Morley's categoricity theorem, proved by Michael D. Morley (1965), states that if a first-order theory in a countable language is categorical in some uncountable cardinality, i.e. all models of this cardinality are isomorphic, then it is categorical in all uncountable cardinalities.

A trivial consequence of the continuum hypothesis is that a complete theory with less than continuum many nonisomorphic countable models can have only countably many. Vaught's conjecture, named after Robert Lawson Vaught, says that this is true even independently of the continuum hypothesis. Many special cases of this conjecture have been established.

Recursion theory

Recursion theory, also called **computability theory**, studies the properties of computable functions and the Turing degrees, which divide the uncomputable functions into sets which have the same level of uncomputability. Recursion theory also includes the study of generalized computability and definability. Recursion theory grew from the work of Alonzo Church and Alan Turing in the 1930s, which was greatly extended by Kleene and Post in the 1940s.

Classical recursion theory focuses on the computability of functions from the natural numbers to the natural numbers. The fundamental results establish a robust, canonical class of computable functions with numerous independent, equivalent characterizations using Turing machines, λ calculus, and other systems. More advanced results concern the structure of the Turing degrees and the lattice of recursively enumerable sets.

Generalized recursion theory extends the ideas of recursion theory to computations that are no longer necessarily finite. It includes the study of computability in higher types as well as areas such as hyperarithmetical theory and

α -recursion theory.

Contemporary research in recursion theory includes the study of applications such as algorithmic randomness and computable model theory as well as new results in pure recursion theory.

Algorithmically unsolvable problems

An important subfield of recursion theory studies algorithmic unsolvability; a decision problem or function problem is **algorithmically unsolvable** if there is no possible computable algorithm which returns the correct answer for all legal inputs to the problem. The first results about unsolvability, obtained independently by Church and Turing in 1936, showed that the Entscheidungsproblem is algorithmically unsolvable. Turing proved this by establishing the unsolvability of the halting problem, a result with far-ranging implications in both recursion theory and computer science.

There are many known examples of undecidable problems from ordinary mathematics. The word problem for groups was proved algorithmically unsolvable by Pyotr Novikov in 1955 and independently by W. Boone in 1959. The busy beaver problem, developed by Tibor Radó in 1962, is another well-known example.

Hilbert's tenth problem asked for an algorithm to determine whether a multivariate polynomial equation with integer coefficients has a solution in the integers. Partial progress was made by Julia Robinson, Martin Davis and Hilary Putnam. The algorithmic unsolvability of the problem was proved by Yuri Matiyasevich in 1970 (Davis 1973).

Proof theory and constructive mathematics

Proof theory is the study of formal proofs in various logical deduction systems. These proofs are represented as formal mathematical objects, facilitating their analysis by mathematical techniques. Several deduction systems are commonly considered, including Hilbert-style deduction systems, systems of natural deduction, and the sequent calculus developed by Gentzen.

The study of **constructive mathematics**, in the context of mathematical logic, includes the study of systems in non-classical logic such as intuitionistic logic, as well as the study of predicative systems. An early proponent of predicativism was Hermann Weyl, who showed it is possible to develop a large part of real analysis using only predicative methods (Weyl 1918).

Because proofs are entirely finitary, whereas truth in a structure is not, it is common for work in constructive mathematics to emphasize provability. The relationship between provability in classical (or nonconstructive) systems and provability in intuitionistic (or constructive, respectively) systems is of particular interest. Results such as the Gödel–Gentzen negative translation show that it is possible to embed (or *translate*) classical logic into intuitionistic logic, allowing some properties about intuitionistic proofs to be transferred back to classical proofs.

Recent developments in proof theory include the study of proof mining by Ulrich Kohlenbach and the study of proof-theoretic ordinals by Michael Rathjen.

Connections with computer science

The study of computability theory in computer science is closely related to the study of computability in mathematical logic. There is a difference of emphasis, however. Computer scientists often focus on concrete programming languages and feasible computability, while researchers in mathematical logic often focus on computability as a theoretical concept and on noncomputability.

The theory of semantics of programming languages is related to model theory, as is program verification (in particular, model checking). The Curry–Howard isomorphism between proofs and programs relates to proof theory, especially intuitionistic logic. Formal calculi such as the lambda calculus and combinatory logic are now studied as idealized programming languages.

Computer science also contributes to mathematics by developing techniques for the automatic checking or even finding of proofs, such as automated theorem proving and logic programming.

Descriptive complexity theory relates logics to computational complexity. The first significant result in this area, Fagin's theorem (1974) established that NP is precisely the set of languages expressible by sentences of existential second-order logic.

Foundations of mathematics

In the 19th century, mathematicians became aware of logical gaps and inconsistencies in their field. It was shown that Euclid's axioms for geometry, which had been taught for centuries as an example of the axiomatic method, were incomplete. The use of infinitesimals, and the very definition of function, came into question in analysis, as pathological examples such as Weierstrass' nowhere-differentiable continuous function were discovered.

Cantor's study of arbitrary infinite sets also drew criticism. Leopold Kronecker famously stated "God made the integers; all else is the work of man," endorsing a return to the study of finite, concrete objects in mathematics. Although Kronecker's argument was carried forward by constructivists in the 20th century, the mathematical community as a whole rejected them. David Hilbert argued in favor of the study of the infinite, saying "No one shall expel us from the Paradise that Cantor has created."

Mathematicians began to search for axiom systems that could be used to formalize large parts of mathematics. In addition to removing ambiguity from previously-naïve terms such as function, it was hoped that this axiomatization would allow for consistency proofs. In the 19th century, the main method of proving the consistency of a set of axioms was to provide a model for it. Thus, for example, non-Euclidean geometry can be proved consistent by defining *point* to mean a point on a fixed sphere and *line* to mean a great circle on the sphere. The resulting structure, a model of elliptic geometry, satisfies the axioms of plane geometry except the parallel postulate.

With the development of formal logic, Hilbert asked whether it would be possible to prove that an axiom system is consistent by analyzing the structure of possible proofs in the system, and showing through this analysis that it is impossible to prove a contradiction. This idea led to the study of proof theory. Moreover, Hilbert proposed that the analysis should be entirely concrete, using the term *finitary* to refer to the methods he would allow but not precisely defining them. This project, known as Hilbert's program, was seriously affected by Gödel's incompleteness theorems, which show that the consistency of formal theories of arithmetic cannot be established using methods formalizable in those theories. Gentzen showed that it is possible to produce a proof of the consistency of arithmetic in a finitary system augmented with axioms of transfinite induction, and the techniques he developed to so do were seminal in proof theory.

A second thread in the history of foundations of mathematics involves nonclassical logics and constructive mathematics. The study of constructive mathematics includes many different programs with various definitions of *constructive*. At the most accommodating end, proofs in ZF set theory that do not use the axiom of choice are called constructive by many mathematicians. More limited versions of constructivism limit themselves to natural numbers, number-theoretic functions, and sets of natural numbers (which can be used to represent real numbers, facilitating the study of mathematical analysis). A common idea is that a concrete means of computing the values of the function must be known before the function itself can be said to exist.

In the early 20th century, Luitzen Egbertus Jan Brouwer founded intuitionism as a philosophy of mathematics. This philosophy, poorly understood at first, stated that in order for a mathematical statement to be true to a mathematician, that person must be able to *intuit* the statement, to not only believe its truth but understand the reason for its truth. A consequence of this definition of truth was the rejection of the law of the excluded middle, for there are statements that, according to Brouwer, could not be claimed to be true while their negations also could not be claimed true. Brouwer's philosophy was influential, and the cause of bitter disputes among prominent mathematicians. Later, Kleene and Kreisel would study formalized versions of intuitionistic logic (Brouwer rejected formalization, and presented his work in unformalized natural language). With the advent of the BHK interpretation

and Kripke models, intuitionism became easier to reconcile with classical mathematics.

See also

- List of mathematical logic topics
- List of computability and complexity topics
- List of set theory topics
- List of first-order theories
- Knowledge representation
- Metalogic

References

Undergraduate texts

- ; Burgess, John; (2002), *Computability and Logic* (4th ed.), Cambridge: Cambridge University Press, ISBN 9780521007580 (pb.).
- Enderton, Herbert (2001), *A mathematical introduction to logic* (2nd ed.), Boston, MA: Academic Press, ISBN 978-0-12-238452-3.
- Hamilton, A.G. (1988), *Logic for Mathematicians* (2nd ed.), Cambridge: Cambridge University Press, ISBN 978-0-521-36865-0.
- Katz, Robert (1964), *Axiomatic Analysis*, Boston, MA: D. C. Heath and Company.
- Mendelson, Elliott (1997), *Introduction to Mathematical Logic* (4th ed.), London: Chapman & Hall, ISBN 978-0-412-80830-2.
- Schwichtenberg, Helmut (2003–2004), *Mathematical Logic* ^[4], Munich, Germany: Mathematisches Institut der Universität München.
- Shawn Hedman, *A first course in logic: an introduction to model theory, proof theory, computability, and complexity*, Oxford University Press, 2004, ISBN 0198529813. Covers logics in close reation with computability theory and complexity theory

Graduate texts

- Andrews, Peter B. (2002), *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof* (2nd ed.), Boston: Kluwer Academic Publishers, ISBN 978-1-4020-0763-7.
- Barwise, Jon, ed. (1982), *Handbook of Mathematical Logic*, Studies in Logic and the Foundations of Mathematics, North Holland, ISBN 978-0-444-86388-1.
- Hodges, Wilfrid (1997), *A shorter model theory*, Cambridge: Cambridge University Press, ISBN 978-0-521-58713-6.
- (2003), *Set Theory: Millennium Edition*, Springer Monographs in Mathematics, Berlin, New York: Springer-Verlag, ISBN 978-3-540-44085-7.
- Shoenfield, Joseph R. (2001) [1967], *Mathematical Logic* (2nd ed.), A K Peters, ISBN 978-1-56881-135-2.
- ; Schwichtenberg, Helmut (2000), *Basic Proof Theory*, Cambridge Tracts in Theoretical Computer Science (2nd ed.), Cambridge: Cambridge University Press, ISBN 978-0-521-77911-1.

Research papers, monographs, texts, and surveys

- Cohen, P. J. (1966), *Set Theory and the Continuum Hypothesis*, Menlo Park, CA: W. A. Benjamin.
- Davis, Martin (1973), "Hilbert's tenth problem is unsolvable" ^[5], *The American Mathematical Monthly* (The American Mathematical Monthly, Vol. 80, No. 3) **80** (3): 233–269, doi:10.2307/2318447, reprinted as an appendix in Martin Davis, *Computability and Unsolvability*, Dover reprint 1982. JStor ^[6]
- Felscher, Walter (2000), "Bolzano, Cauchy, Epsilon, Delta" ^[7], *The American Mathematical Monthly* (The American Mathematical Monthly, Vol. 107, No. 9) **107** (9): 844–862, doi:10.2307/2695743. JSTOR ^[8]
- Ferreirós, José (2001), "The Road to Modern Logic-An Interpretation" ^[9], *Bulletin of Symbolic Logic* (The Bulletin of Symbolic Logic, Vol. 7, No. 4) **7** (4): 441–484, doi:10.2307/2687794. JStor ^[10]
- Hamkins, Joel David; Löwe, Benedikt, "The modal logic of forcing", *Transactions of the American Mathematical Society*, to appear. Electronic posting by the journal ^[11]
- Katz, Victor J. (1998), *A History of Mathematics*, Addison-Wesley, ISBN 0 321 01618 1.
- Morley, Michael (1965), "Categoricity in Power" ^[12], *Transactions of the American Mathematical Society* (Transactions of the American Mathematical Society, Vol. 114, No. 2) **114** (2): 514–538, doi:10.2307/1994188.
- Soare, Robert I. (1996), "Computability and recursion" ^[13], *Bulletin of Symbolic Logic* (The Bulletin of Symbolic Logic, Vol. 2, No. 3) **2** (3): 284–321, doi:10.2307/420992.
- Solovay, Robert M. (1976), "Provability Interpretations of Modal Logic", *Israel Journal of Mathematics* **25**: 287–304, doi:10.1007/BF02757006.
- Woodin, W. Hugh (2001), "The Continuum Hypothesis, Part I", *Notices of the American Mathematical Society* **48** (6). PDF ^[14]

Classical papers, texts, and collections

- Burali-Forti, Cesare (1897), *A question on transfinite numbers*, reprinted in van Heijenoort 1976, pp. 104–111.
- Dedekind, Richard (1872), *Stetigkeit und irrationale Zahlen*. English translation of title: "Consistency and irrational numbers".
- Dedekind, Richard (1888), *Was sind und was sollen die Zahlen?* Two English translations:
 - 1963 (1901). *Essays on the Theory of Numbers*. Beman, W. W., ed. and trans. Dover.
 - 1996. In *From Kant to Hilbert: A Source Book in the Foundations of Mathematics*, 2 vols, Ewald, William B., ed., Oxford University Press: 787–832.
- Fraenkel, Abraham A. (1922), "Der Begriff 'definit' und die Unabhängigkeit des Auswahlaxioms", *Sitzungsberichte der Preussischen Akademie der Wissenschaften, Physikalisch-mathematische Klasse*, pp. 253–257 (German), reprinted in English translation as "The notion of 'definite' and the independence of the axiom of choice", van Heijenoort 1976, pp. 284–289.
- Gentzen, Gerhard (1936), "Die Widerspruchsfreiheit der reinen Zahlentheorie", *Mathematische Annalen* **112**: 132–213, doi:10.1007/BF01565428, reprinted in English translation in Gentzen's *Collected works*, M. E. Szabo, ed., North-Holland, Amsterdam, 1969.
- Gödel, Kurt (1929), "*Über die Vollständigkeit des Logikkalküls*", doctoral dissertation, University Of Vienna. English translation of title: "Completeness of the logical calculus".
- Gödel, Kurt (1930), "Die Vollständigkeit der Axiome des logischen Funktionen-kalküls", *Monatshefte für Mathematik und Physik* **37**: 349–360, doi:10.1007/BF01696781. English translation of title: "The completeness of the axioms of the calculus of logical functions".
- Gödel, Kurt (1931), "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I", *Monatshefte für Mathematik und Physik* **38** (1): 173–198, doi:10.1007/BF01700692, see On Formally Undecidable Propositions of Principia Mathematica and Related Systems for details on English translations.
- Gödel, Kurt (1958), "Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes", *Dialectica. International Journal of Philosophy* **12**: 280–287, doi:10.1111/j.1746-8361.1958.tb01464.x, reprinted in English

- translation in Gödel's *Collected Works*, vol II, Solomon Feferman et al., eds. Oxford University Press, 1990.
- van Heijenoort, Jean, ed. (1967, 1976 3rd printing with corrections), *From Frege to Gödel: A Source Book in Mathematical Logic, 1879-1931* (3rd ed.), Cambridge, Mass: Harvard University Press, ISBN 0-674-32449-8 (pbk.)
 - Hilbert, David (1899), *Grundlagen der Geometrie*, Leipzig: Teubner, English 1902 edition (*The Foundations of Geometry*) republished 1980, Open Court, Chicago.
 - David, Hilbert (1929), "Probleme der Grundlegung der Mathematik", *Mathematische Annalen* **102**: 1–9, doi:10.1007/BF01782335. Lecture given at the International Congress of Mathematicians, 3 September 1928. Published in English translation as "The Grounding of Elementary Number Theory", in Mancosu 1998, pp. 266–273.
 - (1943), "Recursive Predicates and Quantifiers" ^[15], *American Mathematical Society Transactions* (Transactions of the American Mathematical Society, Vol. 53, No. 1) **54** (1): 41–73, doi:10.2307/1990131.
 - Lobachevsky, Nikolai (1840), *Geometrische Untersuchungen zur Theorie der Parallellinien* (German). Reprinted in English translation as "Geometric Investigations on the Theory of Parallel Lines" in *Non-Euclidean Geometry*, Robert Bonola (ed.), Dover, 1955. ISBN 0486600270
 - (1915), "Über Möglichkeiten im Relativkalkül", *Mathematische Annalen* **76**: 447–470, doi:10.1007/BF01458217, ISSN 0025-5831 (German). Translated as "On possibilities in the calculus of relatives" in Jean van Heijenoort, 1967. *A Source Book in Mathematical Logic, 1879–1931*. Harvard Univ. Press: 228–251.
 - Mancosu, Paolo, ed. (1998), *From Brouwer to Hilbert. The Debate on the Foundations of Mathematics in the 1920s*, Oxford: Oxford University Press.
 - Pasch, Moritz (1882), *Vorlesungen über neuere Geometrie*.
 - Peano, Giuseppe (1888), *Arithmetices principia, nova methodo exposita* (Italian), excerpt reprinted in English translation as "The principles of arithmetic, presented by a new method", van Heijenoort 1976, pp. 83–97.
 - Richard, Jules (1905), "Les principes des mathématiques et le problème des ensembles", *Revue générale des sciences pures et appliquées* **16**: 541 (French), reprinted in English translation as "The principles of mathematics and the problems of sets", van Heijenoort 1976, pp. 142–144.
 - Skolem, Thoralf (1920), "Logisch-kombinatorische Untersuchungen über die Erfüllbarkeit oder Beweisbarkeit mathematischer Sätze nebst einem Theoreme über dichte Mengen", *Videnskapsselskapets Skrifter, I. Matematisk-naturvidenskabelig Klasse* **6**: 1–36.
 - Tarski, Alfred (1948), *A decision method for elementary algebra and geometry*, Santa Monica, California: RAND Corporation
 - Turing, Alan M. (1939), "Systems of Logic Based on Ordinals", *Proceedings of the London Mathematical Society* **45** (2): 161–228, doi:10.1112/plms/s2-45.1.161
 - Zermelo, Ernst (1904), "Beweis, daß jede Menge wohlgeordnet werden kann", *Mathematische Annalen* **59**: 514–516, doi:10.1007/BF01445300 (German), reprinted in English translation as "Proof that every set can be well-ordered", van Heijenoort 1976, pp. 139–141.
 - Zermelo, Ernst (1908a), "Neuer Beweis für die Möglichkeit einer Wohlordnung", *Mathematische Annalen* **65**: 107–128, doi:10.1007/BF01450054, ISSN 0025-5831 (German), reprinted in English translation as "A new proof of the possibility of a well-ordering", van Heijenoort 1976, pp. 183–198.
 - Zermelo, Ernst (1908b), "Untersuchungen über die Grundlagen der Mengenlehre", *Mathematische Annalen* **65**: 261–281, doi:10.1007/BF01449999.

External links

- Mathematical Logic around the world ^[16]
- Polyvalued logic ^[17]
- *forall x: an introduction to formal logic* ^[18], by P.D. Magnus, is a free textbook.
- *A Problem Course in Mathematical Logic* ^[19], by Stefan Bilaniuk, is another free textbook.
- Detlovs, Vilnis, and Podnieks, Karlis (University of Latvia) *Introduction to Mathematical Logic*. ^[20] A hyper-textbook.
- Stanford Encyclopedia of Philosophy: Classical Logic ^[21] – by Stewart Shapiro.
- Stanford Encyclopedia of Philosophy: First-order Model Theory ^[22] – by Wilfrid Hodges.
- The London Philosophy Study Guide ^[23] offers many suggestions on what to read, depending on the student's familiarity with the subject:
 - Mathematical Logic ^[24]
 - Set Theory & Further Logic ^[25]
 - Philosophy of Mathematics ^[26]

References

- [1] Undergraduate texts include Boolos, Burgess, and Jeffrey (2002), Enderton (2001), and Mendelson (1997). A classic graduate text by Shoenfield (2001) first appeared in 1967.
- [2] A detailed study of this terminology is given by Soare (1996).
- [3] Ferreirós (2001) surveys the rise of first-order logic over other formal logics in the early 20th century.
- [4] <http://www.mathematik.uni-muenchen.de/~schwicht/lectures/logic/ws03/ml.pdf>
- [5] <http://jstor.org/stable/2318447>
- [6] <http://links.jstor.org/sici?sici=0002-9890%28197303%2980%3A3%3C233%3AHTPIU%3E2.0.CO%3B2-E>
- [7] <http://jstor.org/stable/2695743>
- [8] [http://links.jstor.org/sici?sici=0002-9890\(200011\)107%3A9%3C844%3ABCE%3E2.0.CO%3B2-L](http://links.jstor.org/sici?sici=0002-9890(200011)107%3A9%3C844%3ABCE%3E2.0.CO%3B2-L)
- [9] <http://jstor.org/stable/2687794>
- [10] <http://links.jstor.org/sici?sici=1079-8986%28200112%297%3A4%3C441%3ATRTMLI%3E2.0.CO%3B2-O>
- [11] <http://www.ams.org/tran/0000-000-00/S0002-9947-07-04297-3/home.html>
- [12] <http://jstor.org/stable/1994188>
- [13] <http://jstor.org/stable/420992>
- [14] <http://www.ams.org/notices/200106/fea-woodin.pdf>
- [15] <http://jstor.org/stable/1990131>
- [16] <http://world.logic.at/>
- [17] <http://home.swipnet.se/~w-33552/logic/home/index.htm>
- [18] <http://www.fecundity.com/logic/>
- [19] <http://euclid.trentu.ca/math/sb/pcml/>
- [20] <http://www.ltn.lv/~podnieks/mlog/ml.htm>
- [21] <http://plato.stanford.edu/entries/logic-classical/>
- [22] <http://plato.stanford.edu/entries/modeltheory-fo/>
- [23] <http://www.ucl.ac.uk/philosophy/LPSG/>
- [24] <http://www.ucl.ac.uk/philosophy/LPSG/MathLogic.htm>
- [25] <http://www.ucl.ac.uk/philosophy/LPSG/SetTheory.htm>
- [26] <http://www.ucl.ac.uk/philosophy/LPSG/PhilMath.htm>

Structure (mathematical logic)

In universal algebra and in model theory, a **structure** consists of a set along with a collection of finitary functions and relations which are defined on it.

Universal algebra studies structures that generalize the algebraic structures such as groups, rings, fields, vector spaces and lattices. The term **(universal) algebra** is used for structures with no relation symbols.^[1]

Model theory has a different scope that encompasses more arbitrary theories, including foundational structures such as models of set theory. From the model-theoretic point of view, structures are the objects used to define the semantics of first-order logic.

In database theory, structures with no functions are studied as models for relational databases, in the form of relational models.

Definition

Formally, a **structure** can be defined as a triple $\mathcal{A} = (A, \sigma, I)$ consisting of a **domain** A , a signature σ , and an **interpretation function** I that indicates how the signature is to be interpreted on the domain. To indicate that a structure has a particular signature σ one can refer to it as a σ -structure.

Domain

The domain of a structure is an arbitrary set; it is also called the **underlying set** of the structure, its **carrier** (especially in universal algebra), or its **universe** (especially in model theory). Very often the definition of a structure prohibits the empty domain.^[2]

Sometimes the notation $\text{dom}(\mathcal{A})$ or $|\mathcal{A}|$ is used for the domain of \mathcal{A} , but often no notational distinction is made between a structure and its domain. (I.e. the same symbol \mathcal{A} refers both to the structure and its domain.)^[3]

Signature

The signature of a structure consists of a set of **function symbols** and **relation symbols** along with a function that ascribes to each symbol s a natural number $n = \text{ar}(s)$ which is called the **arity** of s because it is the arity of the interpretation of s .

Since the signatures that arise in algebra often contain only function symbols, a signature with no relation symbols is called an **algebraic signature**. A structure with such a signature is also called an **algebra**; this should not be confused with the notion of an algebra over a field.

Interpretation function

The **interpretation function** I of \mathcal{A} assigns functions and relations to the symbols of the signature. Each function symbol f of arity n is assigned an n -ary function $f^{\mathcal{A}} = I(f)$ on the domain. Each relation symbol R of arity n is assigned an n -ary relation $R^{\mathcal{A}} = I(R) \subseteq A^{\text{ar}(R)}$ on the domain. A nullary function symbol c is called a **constant symbol**, because its interpretation $I(c)$ can be identified with a constant element of the domain.

When a structure (and hence an interpretation function) is given by context, no notational distinction is made between a symbol s and its interpretation $I(s)$. For example if f is a binary function symbol of \mathcal{A} , one simply writes $f : \mathcal{A}^2 \rightarrow \mathcal{A}$ rather than $f^{\mathcal{A}} : |\mathcal{A}|^2 \rightarrow |\mathcal{A}|$.

Examples

The standard signature σ_f for fields consists of two binary function symbols $+$ and \times , a unary function symbol $-$, and the two constant symbols 0 and 1 . Thus a structure (algebra) for this signature consists of a set of elements A together with two binary functions, a unary function, and two distinguished elements; but there is no requirement that it satisfy any of the field axioms. The rational numbers \mathcal{Q} , the real numbers \mathcal{R} and the complex numbers \mathcal{C} , like any other field, can be regarded as σ -structures in an obvious way:

$$\mathcal{Q} = (Q, \sigma_f, I_Q)$$

$$\mathcal{R} = (R, \sigma_f, I_R)$$

$$\mathcal{C} = (C, \sigma_f, I_C)$$

where

$I_Q(+): Q \times Q \rightarrow Q$ is addition of rational numbers,

$I_Q(\times): Q \times Q \rightarrow Q$ is multiplication of rational numbers,

$I_Q(-): Q \rightarrow Q$ is the function that takes each rational number x to $-x$, and

$I_Q(0) \in Q$ is the number 0 and

$I_Q(1) \in Q$ is the number 1 ;

and I_R and I_C are similarly defined.

But the ring \mathcal{Z} of integers, which is not a field, is also a σ_f -structure in the same way. In fact, there is no requirement that *any* of the field axioms hold in a σ_f -structure.

A signature for ordered fields needs an additional binary relation such as $<$ or \leq , and therefore structures for such a signature are not algebras, even though they are of course algebraic structures in the usual, loose sense of the word.

The ordinary signature for set theory includes a single binary relation \in . A structure for this signature consists of a set of elements and an interpretation of the \in relation as a binary relation on these elements.

Induced substructures and closed subsets

\mathcal{A} is called an (induced) substructure of \mathcal{B} if

- \mathcal{A} and \mathcal{B} have the same signature $\sigma(\mathcal{A}) = \sigma(\mathcal{B})$;
- the domain of \mathcal{A} is contained in the domain of \mathcal{B} : $|\mathcal{A}| \subseteq |\mathcal{B}|$; and
- the interpretations of all function and relation symbols agree on $|\mathcal{B}|$.

The usual notation for this relation is $\mathcal{A} \subseteq \mathcal{B}$.

A subset $B \subseteq |\mathcal{A}|$ of the domain of a structure \mathcal{A} is called **closed** if it is closed under the functions of \mathcal{A} , i.e. if the following condition is satisfied: for every natural number n , every n -ary function symbol f (in the signature of \mathcal{A}) and all elements $b_1, b_2, \dots, b_n \in B$, the result of applying f to the n -tuple $b_1 b_2 \dots b_n$ is again an element of B : $f(b_1, b_2, \dots, b_n) \in B$.

For every subset $B \subseteq |\mathcal{A}|$ there is a smallest closed subset of $|\mathcal{A}|$ that contains B . It is called the closed subset **generated** by B , or the **hull** of B , and denoted by $\langle B \rangle$ or $\langle B \rangle_{\mathcal{A}}$. The operator $\langle \rangle$ is a finitary closure operator on the set of subsets of $|\mathcal{A}|$.

If $\mathcal{A} = (A, \sigma, I)$ and $B \subseteq A$ is a closed subset, then (B, σ, I') is an induced substructure of \mathcal{A} , where I' assigns to every symbol of σ the restriction to B of its interpretation in \mathcal{A} . Conversely, the domain of an induced substructure is a closed subset.

The closed subsets (or induced substructures) of a structure form a lattice. The meet of two subsets is their intersection. The join of two subsets is the closed subset generated by their union. Universal algebra studies the lattice of substructures of a structure in detail.

Examples

Let $\sigma = \{+, \times, -, 0, 1\}$ be again the standard signature for fields. When regarded as σ -structures in the natural way, the rational numbers form a substructure of the real numbers, and the real numbers form a substructure of the complex numbers. The rational numbers are the smallest substructure of the real (or complex) numbers that also satisfies the field axioms.

The set of integers gives an even smaller substructure of the real numbers which is not a field. Indeed, the integers are the substructure of the real numbers generated by the empty set, using this signature. The notion in abstract algebra that corresponds to a substructure of a field, in this signature, is that of a subring, rather than that of a subfield.

The most obvious way to represent a graph as a structure is with a signature σ consisting of a single binary relation symbol E . The vertices of the graph form the domain of the structure, and for two vertices a and b , $(a, b) \in E$ means that a and b are connected by an edge. In this encoding, the notion of induced substructure is more restrictive than the notion of subgraph. For example, let G be a graph consisting of two vertices connected by an edge, and let H be the graph consisting of the same vertices but no edges. H is a subgraph of G , but not an induced substructure. The notion in graph theory that corresponds to induced substructures is that of induced subgraphs.

Homomorphisms and embeddings

Homomorphisms

Given two structures \mathcal{A} and \mathcal{B} of the same signature σ , a **(σ)-homomorphism** from \mathcal{A} to \mathcal{B} is a map $h : |\mathcal{A}| \rightarrow |\mathcal{B}|$ which preserves the functions and relations. More precisely:

- For every n -ary function symbol f of σ and any elements $a_1, a_2, \dots, a_n \in |\mathcal{A}|$, the following equation holds:

$$h(f(a_1, a_2, \dots, a_n)) = f(h(a_1), h(a_2), \dots, h(a_n)).$$

- For every n -ary relation symbol R of σ and any elements $a_1, a_2, \dots, a_n \in |\mathcal{A}|$, the following implication holds:

$$(a_1, a_2, \dots, a_n) \in R \implies (h(a_1), h(a_2), \dots, h(a_n)) \in R.$$

The notation for a homomorphism h from \mathcal{A} to \mathcal{B} is $h : \mathcal{A} \rightarrow \mathcal{B}$.

For every signature σ there is a concrete category $\sigma\text{-Hom}$ which has σ -structures as objects and σ -homomorphisms as morphisms.

A homomorphism $h : \mathcal{A} \rightarrow \mathcal{B}$ is sometimes called **strong** if for every n -ary relation symbol R and any elements $b_1, b_2, \dots, b_n \in |\mathcal{B}|$ such that $(b_1, b_2, \dots, b_n) \in R$, there are $a_1, a_2, \dots, a_n \in |\mathcal{A}|$ such that $(a_1, a_2, \dots, a_n) \in R$ and $b_1 = h(a_1)$, $b_2 = h(a_2)$, \dots , $b_n = h(a_n)$. The strong homomorphisms give rise to a subcategory of $\sigma\text{-Hom}$.

Embeddings

A (σ)-homomorphism $h : \mathcal{A} \rightarrow \mathcal{B}$ is called a **(σ)-embedding** if it is one-to-one and

- for every n -ary relation symbol R of σ and any elements a_1, a_2, \dots, a_n , the following equivalence holds:

$$(a_1, a_2, \dots, a_n) \in R \iff (h(a_1), h(a_2), \dots, h(a_n)) \in R.$$

Thus an embedding is the same thing as a strong homomorphism which is one-to-one. The category $\sigma\text{-Emb}$ of σ -structures and σ -embeddings is a concrete subcategory of $\sigma\text{-Hom}$.

Induced substructures correspond to subobjects in $\sigma\text{-Emb}$. If σ has only function symbols, $\sigma\text{-Emb}$ is the subcategory of monomorphisms of $\sigma\text{-Hom}$. In this case induced substructures also correspond to subobjects in $\sigma\text{-Hom}$.

Example

As seen above, in the standard encoding of graphs as structures the induced substructures are precisely the induced subgraphs. However, a homomorphism between graphs is the same thing as a homomorphism between the two structures coding the graph. In the example of the previous section, even though the subgraph H of G is not induced, the identity map $\text{id}: H \rightarrow G$ is a homomorphism. This map is in fact a monomorphism in the category $\sigma\text{-Hom}$, and therefore H is a subobject of G which is not an induced substructure.

Homomorphism problem

The following problem is known as the *homomorphism problem*:

Given two finite structures \mathcal{A} and \mathcal{B} of a finite relational signature, find a homomorphism $h: \mathcal{A} \rightarrow \mathcal{B}$ or show that no such homomorphism exists.

Every constraint satisfaction problem (CSP) has a translation into the homomorphism problem.^[4] Therefore the complexity of CSP can be studied using the methods of finite model theory.

Another application is in database theory, where a relational model of a database is essentially the same thing as a relational structure. It turns out that a conjunctive query on a database can be described by another structure in the same signature as the database model. A homomorphism from the relational model to the structure representing the query is the same thing as a solution to the query. This shows that the conjunctive query problem is also equivalent to the homomorphism problem.

Structures and first-order logic

Structures are sometimes referred to as "first-order structures". This is misleading, as nothing in their definition ties them to any specific logic, and in fact they are suitable as semantic objects both for very restricted fragments of first-order logic such as that used in universal algebra, and for second-order logic. In connection with first-order logic and model theory, structures are often called **models**, even when the question "models of what?" has no obvious answer.

Satisfaction relation

Each first-order structure \mathcal{M} has a **satisfaction relation** $\mathcal{M} \models \phi$ defined for all formulas ϕ in the language consisting of the language of \mathcal{M} together with a constant symbol for each element of M , which is interpreted as that element. This relation is defined inductively using Tarski's T-schema.

A structure \mathcal{M} is said to be a **model** of a theory T if the language of \mathcal{M} is the same as the language of T and every sentence in T is satisfied by \mathcal{M} . Thus, for example, a "ring" is a structure for the language of rings that satisfies each of the ring axioms, and a model of ZFC set theory is a structure in the language of set theory that satisfies each of the ZFC axioms.

Definable relations

An n -ary relation R on the universe M of a structure \mathcal{M} is said to be **definable** (or **explicitly definable**, or **\emptyset -definable**) if there is a formula $\varphi(x_1, \dots, x_n)$ such that

$$R = \{(a_1, \dots, a_n) \in M^n : \mathcal{M} \models \varphi(a_1, \dots, a_n)\}.$$

In other words, R is definable if and only if there is a formula φ such that

$$(a_1, \dots, a_n) \in R \Leftrightarrow \mathcal{M} \models \varphi(a_1, \dots, a_n)$$

is correct.

An important special case is the definability of specific elements. An element m of M is definable in \mathcal{M} if and only if there is a formula $\varphi(x)$ such that

$$\mathcal{M} \models \forall x(x = m \leftrightarrow \phi(x)).$$

Definability with parameters

A relation R is said to be **definable with parameters** (or $|\mathcal{M}|$ -**definable**) if there is a formula φ with parameters from \mathcal{M} such that R is definable using φ . Every element of a structure is definable using the element itself as a parameter.

Implicit definability

Recall from above that an n -ary relation R on the universe M of a structure \mathcal{M} is explicitly definable if there is a formula $\varphi(x_1, \dots, x_n)$ such that

$$R = \{(a_1, \dots, a_n) \in M^n : \mathcal{M} \models \varphi(a_1, \dots, a_n)\}$$

Here the formula φ used to define a relation R must be over the signature of \mathcal{M} and so φ may not mention R itself, since R is not in the signature of \mathcal{M} . If there is a formula φ in the extended language containing the language of \mathcal{M} and a new symbol R , and the relation R is the only relation on \mathcal{M} such that $\mathcal{M} \models \varphi$, then R is said to be **implicitly definable** over \mathcal{M} .

There are many examples of implicitly definable relations that are not explicitly definable.

Many-sorted structures

Structures as defined above are sometimes called **one-sorted structures** to distinguish them from the more general **many-sorted structures**. A many-sorted structure can have an arbitrary number of domains. The **sorts** are part of the signature, and they play the role of names for the different domains. Many-sorted signatures also prescribe on which sorts the functions and relations of a many-sorted structure are defined. Therefore the arities of function symbols or relation symbols must be more complicated objects such as tuples of sorts rather than natural numbers.

Vector spaces, for example, can be regarded as two-sorted structures in the following way. The two-sorted signature of vector spaces consists of two sorts V (for vectors) and S (for scalars) and the following function symbols:

- $+$ _{S} and \times _{S} of arity $(S, S; S)$.
- $+$ _{V} of arity $(V, V; V)$.
- \times of arity $(S, V; V)$.
- $-$ _{S} of arity $(S; S)$.
- $-$ _{V} of arity $(V; V)$.
- 0 _{S} and 1 _{S} of arity (S) .
- 0 _{V} of arity (V) .

If V is a vector space over a field F , the corresponding two-sorted structure \mathcal{V} consists of the vector domain $|\mathcal{V}|_V = V$, the scalar domain $|\mathcal{V}|_S = F$, and the obvious functions, such as the vector zero $0_V^\mathcal{V} = 0 \in |\mathcal{V}|_V$, the scalar zero $0_S^\mathcal{V} = 0 \in |\mathcal{V}|_S$, or scalar multiplication $\times^\mathcal{V} : |\mathcal{V}|_S \times |\mathcal{V}|_V \rightarrow |\mathcal{V}|_V$.

Many-sorted structures are often used as a convenient tool even when they could be avoided with a little effort. But they are rarely defined in a rigorous way, because it is straightforward and tedious (hence unrewarding) to carry out the generalization explicitly.

In most mathematical endeavours, not much attention is paid to the sorts. A many-sorted logic however naturally leads to a type theory. As Bart Jacobs puts it: "A logic is always a logic over a type theory." This emphasis in turn leads to categorical logic because a logic over a type theory categorically corresponds to one ("total") category, capturing the logic, being fibred over another ("base") category, capturing the type theory.^[5]

Other generalizations

Partial algebras

Both universal algebra and model theory study classes of (structures or) algebras that are defined by a signature and a set of axioms. In the case of model theory these axioms have the form of first-order sentences. The formalism of universal algebra is much more restrictive; essentially it only allows first-order sentences that have the form of universally quantified equations between terms, e.g. $\forall x \forall y (x + y = y + x)$. One consequence is that the choice of a signature is more significant in universal algebra than it is in model theory. For example the class of groups, in the signature consisting of the binary function symbol \times and the constant symbol 1 , is an elementary class, but it is not a variety. Universal algebra solves this problem by adding a unary function symbol $^{-1}$.

In the case of fields this strategy works only for addition. For multiplication it fails because 0 does not have a multiplicative inverse. An ad hoc attempt to deal with this would be to define $0^{-1} = 0$. (This attempt fails, essentially because with this definition $0 \times 0^{-1} = 1$ is not true.) Therefore one is naturally led to allow partial functions, i.e., functions which are defined only on a subset of their domain. However, there are several obvious ways to generalize notions such as substructure, homomorphism and identity.

Structures for typed languages

In type theory, there are many sorts of variables, each of which has a **type**. Types are inductively defined; given two types δ and σ there is also a type $\sigma \rightarrow \delta$ that represents functions from objects of type σ to objects of type δ . A structure for a typed language (in the ordinary first-order semantics) must include a separate set of objects of each type, and for a function type the structure must have complete information about the function represented by each object of that type.

Higher-order languages

There is more than one possible semantics for higher-order logic, as discussed in the article on second-order logic. When using full higher-order semantics, a structure need only have a universe for objects of type 0 , and the T-schema is extended so that a quantifier over a higher-order type is satisfied by the model if and only if it is disquotationally true. When using first-order semantics, an additional sort is added for each higher-order type, as in the case of a many sorted first order language.

Structures that are proper classes

In the study of set theory and category theory, it is sometimes useful to consider structures in which the domain of discourse is a proper class instead of a set. These structures are sometimes called **class models** to distinguish them from the "set models" discussed above. When the domain is a proper class, each function and relation symbol may also be represented by a proper class.

In Bertrand Russell's *Principia Mathematica*, structures were also allowed to have a proper class as their domain.

References

- Burris, Stanley N.; Sankappanavar, H. P. (1981), *A Course in Universal Algebra* ^[1], Berlin, New York: Springer-Verlag
- Chang, Chen Chung; Keisler, H. Jerome (1989) [1973], *Model Theory*, Elsevier, ISBN 978-0-7204-0692-4
- Diestel, Reinhard (2005) [1997], *Graph Theory* ^[6], Graduate Texts in Mathematics, **173** (3rd ed.), Berlin, New York: Springer-Verlag, ISBN 978-3-540-26183-4
- Ebbinghaus, Heinz-Dieter; Flum, Jörg; Thomas, Wolfgang (1994), *Mathematical Logic* (2nd ed.), New York: Springer, ISBN 978-0-387-94258-2
- Hinman, P. (2005), *Fundamentals of Mathematical Logic*, A K Peters, ISBN 978-1-56881-262-5
- Hodges, Wilfrid (1993), *Model theory*, Cambridge: Cambridge University Press, ISBN 978-0-521-30442-9
- Hodges, Wilfrid (1997), *A shorter model theory*, Cambridge: Cambridge University Press, ISBN 978-0-521-58713-6
- Marker, David (2002), *Model Theory: An Introduction*, Berlin, New York: Springer-Verlag, ISBN 978-0-387-98760-6
- Poizat, Bruno (2000), *A Course in Model Theory: An Introduction to Contemporary Mathematical Logic*, Berlin, New York: Springer-Verlag, ISBN 978-0-387-98655-5
- Rothmaler, Philipp (2000), *Introduction to Model Theory*, London: CRC Press, ISBN 9789056993139

External links

- Semantics ^[7] section in Classical Logic ^[21] (an entry of Stanford Encyclopedia of Philosophy ^[8])

References

- [1] Some authors refer to structures as "algebras" when generalizing universal algebra to allow relations as well as functions.
- [2] This is similar to the definition of a prime number in elementary number theory, which has been carefully chosen so that the irreducible number 1 is not considered prime. The convention that structures may not be empty is particularly important in logic, because several common inference rules are not sound when empty structures are permitted.
- [3] As a consequence of these conventions, the notation $|\mathcal{A}|$ may also be used to refer to the cardinality of the domain of \mathcal{A} . In practice this never leads to confusion.
- [4] Jeavons, Peter; David Cohen; Justin Pearson (1998), "Constraints and universal algebra", *Annals of Mathematics and Artificial Intelligence* **24**: 51–67, doi:10.1023/A:1018941030227.
- [5] Jacobs, Bart (1999), *Categorical Logic and Type Theory*, Elsevier, pp. 1–4
- [6] <http://www.math.uni-hamburg.de/home/diestel/books/graph.theory/>
- [7] <http://plato.stanford.edu/entries/logic-classical/#4>
- [8] <http://plato.stanford.edu>

Universal algebra

Universal algebra (sometimes called **general algebra**) is the field of mathematics that studies algebraic structures themselves, not examples ("models") of algebraic structures. For instance, rather than take particular groups as the object of study, in universal algebra one takes "the theory of groups" as an object of study.

Basic idea

From the point of view of universal algebra, an **algebra** (or **algebraic structure**) is a set A together with a collection of operations on A . An **n -ary operation** on A is a function that takes n elements of A and returns a single element of A . Thus, a 0-ary operation (or *nullary operation*) can be represented simply as an element of A , or a *constant*, often denoted by a letter like a . A 1-ary operation (or *unary operation*) is simply a function from A to A , often denoted by a symbol placed in front of its argument, like $\sim x$. A 2-ary operation (or *binary operation*) is often denoted by a symbol placed between its arguments, like $x * y$. Operations of higher or unspecified arity are usually denoted by function symbols, with the arguments placed in parentheses and separated by commas, like $f(x,y,z)$ or $f(x_1, \dots, x_n)$. Some researchers allow infinitary operations, such as $\bigwedge_{\alpha \in J} x_\alpha$ where J is an infinite index set, thus leading into the algebraic theory of complete lattices. One way of talking about an algebra, then, is by referring to it as an algebra of a certain type Ω , where Ω is an ordered sequence of natural numbers representing the arity of the operations of the algebra.

Equations

After the operations have been specified, the nature of the algebra can be further limited by axioms, which in universal algebra often take the form of identities, or **equational laws**. An example is the associative axiom for a binary operation, which is given by the equation $x * (y * z) = (x * y) * z$. The axiom is intended to hold for all elements x , y , and z of the set A .

Varieties

An algebraic structure which can be defined by identities is called a **variety**, and these are sufficiently important that some authors consider varieties the only object of study in universal algebra, while others consider them an object.

Restricting one's study to varieties rules out:

- Predicate logic, notably quantification, including existential quantification (\exists) and universal quantification (\forall)
- Relations, including inequalities, both $a \neq b$ and order relations

In this narrower definition, universal algebra can be seen as a special branch of model theory, in which we are typically dealing with structures having operations only (i.e. the type can have symbols for functions but not for relations other than equality), and in which the language used to talk about these structures uses equations only.

Not all algebraic structures in a wider sense fall into this scope. For example ordered groups are not studied in mainstream universal algebra because they involve an ordering relation.

A more fundamental restriction is that universal algebra cannot study the class of fields, because there is no type in which all field laws can be written as equations (inverses of elements are defined for all *non-zero* elements in a field, so inversion cannot simply be added to the type).

One advantage of this restriction is that the structures studied in universal algebra can be defined in any category which has *finite products*. For example, a topological group is just a group in the category of topological spaces.

Examples

Most of the usual algebraic systems of mathematics are examples of varieties, but not always in an obvious way – the usual definitions often involve quantification or inequalities.

Groups

To see how this works, let's consider the definition of a group. Normally a group is defined in terms of a single binary operation $*$, subject to these axioms:

- Associativity (as in the previous section): $x * (y * z) = (x * y) * z$.
- Identity element: There exists an element e such that for each element x , $e * x = x = x * e$.
- Inverse element: It can easily be seen that the identity element is unique. If we denote this unique identity element by e then for each x , there exists an element i such that $x * i = e = i * x$.

(Sometimes you will also see an axiom called "closure", stating that $x * y$ belongs to the set A whenever x and y do. But from a universal algebraist's point of view, that is already implied when you call $*$ a binary operation.)

Now, this definition of a group is problematic from the point of view of universal algebra. The reason is that the axioms of the identity element and inversion are not stated purely in terms of equational laws but also have clauses involving the phrase "there exists ... such that ...". This is inconvenient; the list of group properties can be simplified to universally quantified equations if we add a nullary operation e and a unary operation \sim in addition to the binary operation $*$, then list the axioms for these three operations as follows:

- Associativity: $x * (y * z) = (x * y) * z$.
- Identity element: $e * x = x = x * e$.
- Inverse element: $x * (\sim x) = e = (\sim x) * x$.

(Of course, we usually write " x^{-1} " instead of " $\sim x$ ", which shows that the notation for operations of low arity is not *always* as given in the second paragraph.)

What has changed is that in the usual definition there are:

- a single binary operation (signature (2))
- 1 equational law (associativity)
- 2 quantified laws (identity and inverse)

...while in the universal algebra definition there are

- 3 operations: one binary, one unary, and one nullary (signature (2,1,0))
- 3 equational laws (associativity, identity, and inverse)
- no quantified laws

It's important to check that this really does capture the definition of a group. The reason that it might not is that specifying one of these universal groups might give more information than specifying one of the usual kind of group. After all, nothing in the usual definition said that the identity element e was *unique*; if there is another identity element e' , then it's ambiguous which one should be the value of the nullary operator e . However, this is not a problem because identity elements can be proved to be always unique. The same thing is true of inverse elements. So the universal algebraist's definition of a group really is equivalent to the usual definition.

Basic constructions

We assume that the type, Ω , has been fixed. Then there are three basic constructions in universal algebra: homomorphic image, subalgebra, and product.

A homomorphism between two algebras A and B is a function $h: A \rightarrow B$ from the set A to the set B such that, for every operation f (of arity, say, n), $h(f_A(x_1, \dots, x_n)) = f_B(h(x_1), \dots, h(x_n))$. (Here, subscripts are placed on f to indicate whether it is the version of f in A or B . In theory, you could tell this from the context, so these subscripts are usually left off.) For example, if e is a constant (nullary operation), then $h(e_A) = e_B$. If \sim is a unary operation, then $h(\sim x) = \sim h(x)$. If $*$ is a binary operation, then $h(x * y) = h(x) * h(y)$. And so on. A few of the things that can be done with homomorphisms, as well as definitions of certain special kinds of homomorphisms, are listed under the entry Homomorphism. In particular, we can take the homomorphic image of an algebra, $h(A)$.

A subalgebra of A is a subset of A that is closed under all the operations of A . A product of some set of algebraic structures is the cartesian product of the sets with the operations defined coordinatewise.

Some basic theorems

- The Isomorphism theorems, which encompass the isomorphism theorems of groups, rings, modules, etc.
- Birkhoff's HSP Theorem, which states that a class of algebras is a variety if and only if it is closed under homomorphic images, subalgebras, and arbitrary direct products.

Motivations and applications

In addition to its unifying approach, universal algebra also gives deep theorems and important examples and counterexamples. It provides a useful framework for those who intend to start the study of new classes of algebras. It can enable the use of methods invented for some particular classes of algebras to other classes of algebras, by recasting the methods in terms of universal algebra (if possible), and then interpreting these as applied to other classes. It has also provided conceptual clarification; as J.D.H. Smith puts it, *"What looks messy and complicated in a particular framework may turn out to be simple and obvious in the proper general one."*

In particular, universal algebra can be applied to the study of monoids, rings, and lattices. Before universal algebra came along, many theorems (most notably the isomorphism theorems) were proved separately in all of these fields, but with universal algebra, they can be proven once and for all for every kind of algebraic system.

Category theory and operads

A more generalised programme along these lines is carried out by category theory. Given a list of operations and axioms in universal algebra, the corresponding algebras and homomorphisms are the objects and morphisms of a category. Category theory applies to many situations where universal algebra does not, extending the reach of the theorems. Conversely, many theorems that hold in universal algebra do not generalise all the way to category theory. Thus both fields of study are useful.

A more recent development in category theory that generalizes operations is operad theory – an operad is a set of operations, similar to a universal algebra.

History

In Alfred North Whitehead's book *A Treatise on Universal Algebra*, published in 1898, the term *universal algebra* had essentially the same meaning that it has today. Whitehead credits William Rowan Hamilton and Augustus De Morgan as originators of the subject matter, and James Joseph Sylvester with coining the term itself^[1].

At the time structures such as Lie algebras and hyperbolic quaternions drew attention to the need to expand algebraic structures beyond the associatively multiplicative class. In a review Alexander Macfarlane wrote: "The main idea of the work is not unification of the several methods, nor generalization of ordinary algebra so as to include them, but rather the comparative study of their several structures." At the time George Boole's algebra of logic made a strong counterpoint to ordinary number algebra, so the term "universal" served to calm strained sensibilities.

Whitehead's early work sought to unify quaternions (due to Hamilton), Grassmann's *Ausdehnungslehre*, and Boole's algebra of logic. Whitehead wrote in his book:

"Such algebras have an intrinsic value for separate detailed study; also they are worthy of comparative study, for the sake of the light thereby thrown on the general theory of symbolic reasoning, and on algebraic symbolism in particular. The comparative study necessarily presupposes some previous separate study, comparison being impossible without knowledge."^[2]

Whitehead, however, had no results of a general nature. Work on the subject was minimal until the early 1930s, when Garrett Birkhoff and Øystein Ore began publishing on universal algebras. Developments in metamathematics and category theory in the 1940s and 1950s furthered the field, particularly the work of Abraham Robinson, Alfred Tarski, Andrzej Mostowski, and their students (Brainerd 1967).

In the period between 1935 and 1950, most papers were written along the lines suggested by Birkhoff's papers, dealing with free algebras, congruence and subalgebra lattices, and homomorphism theorems. Although the development of mathematical logic had made applications to algebra possible, they came about slowly; results published by Anatoly Maltsev in the 1940s went unnoticed because of the war. Tarski's lecture at the 1950 International Congress of Mathematicians in Cambridge ushered in a new period in which model-theoretic aspects were developed, mainly by Tarski himself, as well as C.C. Chang, Leon Henkin, Bjarni Jónsson, R. C. Lyndon, and others.

In the late 1950s, E. Marczewski^[3] emphasized the importance of free algebras, leading to the publication of more than 50 papers on the algebraic theory of free algebras by Marczewski himself, together with J. Mycielski, W. Narkiewicz, W. Nitka, J. Płonka, S. Świerczkowski, K. Urbanik, and others.

See also

- category theory
- graph algebra
- homomorphism
- lattice theory
- signature
- variety
- clone
- operad theory
- model theory
- Universal algebraic geometry

References

- Bergman, George M., 1998. *An Invitation to General Algebra and Universal Constructions* ^[4] (pub. Henry Helson, 15 the Crescent, Berkeley CA, 94708) 398 pp. ISBN 0-9655211-4-1.
- Birkhoff, Garrett, 1946. Universal algebra. *Comptes Rendus du Premier Congrès Canadien de Mathématiques*, University of Toronto Press, Toronto, pp. 310–326.
- Brainerd, Barron, Aug-Sep 1967. Review of *Universal Algebra* by P. M. Cohn. *American Mathematical Monthly*, 74(7): 878-880.
- Burris, Stanley N., and H.P. Sankappanavar, 1981. *A Course in Universal Algebra* ^[1] Springer-Verlag. ISBN 3-540-90578-2 *Free online edition*.
- Cohn, Paul Moritz, 1981. *Universal Algebra*. Dordrecht , Netherlands: D.Reidel Publishing. ISBN 90-277-1213-1 (*First published in 1965 by Harper & Row*)
- Freese, Ralph, and Ralph McKenzie, 1987. *Commutator Theory for Congruence Modular Varieties* ^[5], 1st ed. *London Mathematical Society Lecture Note Series*, 125. Cambridge Univ. Press. ISBN 0-521-34832-3. *Free online second edition*.
- Grätzer, George, 1968. *Universal Algebra* D. Van Nostrand Company, Inc.
- Hobby, David, and Ralph McKenzie, 1988. *The Structure of Finite Algebras* ^[6] American Mathematical Society. ISBN 0-8218-3400-2. *Free online edition*.
- Jipsen, Peter, and Henry Rose, 1992. *Varieties of Lattices* ^[7], Lecture Notes in Mathematics 1533. Springer Verlag. ISBN 0-387-56314-8. *Free online edition*.
- Pigozzi, Don. *General Theory of Algebras* ^[8].
- Smith, J.D.H., 1976. *Mal'cev Varieties*, Springer-Verlag.
- Whitehead, Alfred North, 1898. *A Treatise on Universal Algebra* ^[9], Cambridge. (*Mainly of historical interest.*)

External links

- *Algebra Universalis* ^[10]—a journal dedicated to Universal Algebra.

References

- [1] Grätzer, George. **Universal Algebra**, Van Nostrand Co., Inc., 1968, p. v.
- [2] Quoted in Grätzer, George. **Universal Algebra**, Van Nostrand Co., Inc., 1968.
- [3] Marczewski, E. "A general scheme of the notions of independence in mathematics." *Bull. Acad. Polon. Sci. Ser. Sci. Math. Astronom. Phys.* **6** (1958), 731-736.
- [4] <http://math.berkeley.edu/~gbergman/245/>
- [5] <http://www.math.hawaii.edu/~ralph/Commutator>
- [6] http://www.ams.org/online_bks/conm76
- [7] <http://www1.chapman.edu/~jipsen/JipsenRoseVoL.html>
- [8] <http://bigcheese.math.sc.edu/~mcnulty/alglatvar/pigozzinotes.pdf>
- [9] <http://historical.library.cornell.edu/cgi-bin/cul.math/docviewer?did=01950001&seq=5>
- [10] <http://www.birkhauser.ch/AU>

Model theory

In mathematics, **model theory** is the study of (classes of) mathematical structures such as groups, fields, graphs, or even universes of set theory, using tools from mathematical logic. A structure that gives meaning to the sentences of a formal language is called a model *for* the language. If a model for a language moreover satisfies a particular sentence or theory (set of sentences), it is called a model *of* the sentence or theory. Model theory has close ties to algebra and universal algebra.

This article focuses on finitary first order model theory of infinite structures. Finite model theory, which concentrates on finite structures, diverges significantly from the study of infinite structures in both the problems studied and the techniques used. Model theory in higher-order logics or infinitary logics is hampered by the fact that completeness does not in general hold for these logics. However, a great deal of study has also been done in such languages.

Introduction

Model theory recognises and is intimately concerned with a duality: It examines semantical elements by means of syntactical elements of a corresponding language. To quote the first page of Chang and Keisler (1990):

universal algebra + logic = **model theory**.

In a similar way to proof theory, model theory is situated in an area of interdisciplinarity between mathematics, philosophy, and computer science. The most important professional organization in the field of model theory is the Association for Symbolic Logic.

An incomplete and somewhat arbitrary subdivision of model theory is into classical model theory, model theory applied to groups and fields, and geometric model theory. A missing subdivision is computable model theory, but this can arguably be viewed as an independent subfield of logic. Examples of early theorems from classical model theory include Gödel's completeness theorem, the upward and downward Löwenheim–Skolem theorems, Vaught's two-cardinal theorem, Scott's isomorphism theorem, the omitting types theorem, and the Ryll-Nardzewski theorem. Examples of early results from model theory applied to fields are Tarski's elimination of quantifiers for real closed fields, Ax's theorem on pseudo-finite fields, and Robinson's development of nonstandard analysis. An important step in the evolution of classical model theory occurred with the birth of stability theory (through Morley's theorem on uncountably categorical theories and Shelah's classification program), which developed a calculus of independence and rank based on syntactical conditions satisfied by theories. During the last several decades applied model theory has repeatedly merged with the more pure stability theory. The result of this synthesis is called geometric model theory in this article (which is taken to include o-minimality, for example, as well as classical geometric stability theory). An example of a theorem from geometric model theory is Hrushovski's proof of the Mordell–Lang conjecture for function fields. The ambition of geometric model theory is to provide a *geography of mathematics* by embarking on a detailed study of definable sets in various mathematical structures, aided by the substantial tools developed in the study of pure model theory.

Example

To illustrate the basic relationship involving syntax and semantics in the context of a non-trivial models, one can start, on the syntactic side, with suitable axioms for the natural numbers such as Peano axioms, and the associated theory. Going on to the semantic side, one has the usual counting numbers as a model. In the 1930s, Skolem developed alternative models satisfying the axioms. This illustrates what is meant by interpreting a language or theory in a particular model. A more traditional example is interpreting the axioms of a particular algebraic system such as a group, in the context of a model provided by a specific group.

Universal algebra

Fundamental concepts in universal algebra are signatures σ and σ -algebras. Since these concepts are formally defined in the article on structures, the present article can content itself with an informal introduction which consists in examples of how these terms are used.

The standard signature of rings is $\sigma_{\text{ring}} = \{\times, +, -, 0, 1\}$, where \times and $+$ are binary, $-$ is unary, and 0 and 1 are nullary.

The standard signature of semirings is $\sigma_{\text{smr}} = \{\times, +, 0, 1\}$, where the arities are as above.

The standard signature of (multiplicative) groups is $\sigma_{\text{grp}} = \{\times, ^{-1}, 1\}$, where \times is binary, $^{-1}$ is unary and 1 is nullary.

The standard signature of monoids is $\sigma_{\text{mnd}} = \{\times, 1\}$.

A ring is a σ_{ring} -structure which satisfies the identities $u + (v + w) = (u + v) + w$, $u + v = v + u$, $u + 0 = u$, $u + (-u) = 0$, $u \times (v \times w) = (u \times v) \times w$, $u \times 1 = u$, $1 \times u = u$, $u \times (v + w) = (u \times v) + (u \times w)$ and $(v + w) \times u = (v \times u) + (w \times u)$.

A group is a σ_{grp} -structure which satisfies the identities $u \times (v \times w) = (u \times v) \times w$, $u \times 1 = u$, $1 \times u = u$, $u \times u^{-1} = 1$ and $u^{-1} \times u = 1$.

A monoid is a σ_{mnd} -structure which satisfies the identities $u \times (v \times w) = (u \times v) \times w$, $u \times 1 = u$ and $1 \times u = u$.

A semigroup is a σ_{mnd} -structure which satisfies the identity $u \times (v \times w) = (u \times v) \times w$.

A magma is just a $\{\times\}$ -structure.

This is a very efficient way to define most classes of algebraic structures, because there is also the concept of σ -homomorphism, which correctly specializes to the usual notions of homomorphism for groups, semigroups, magmas and rings. For this to work, the signature must be chosen well.

Terms such as the σ_{ring} -term $t(u, v, w)$ given by $(u + (v \times w)) + (-1)$ are used to define identities $t = t'$, but also to construct free algebras. An equational class is a class of structures which, like the examples above and many others, is defined as the class of all σ -structures which satisfy a certain set of identities. Birkhoff's theorem states:

A class of σ -structures is an equational class if and only if it is not empty and closed under subalgebras, homomorphic images, and direct products.

An important non-trivial tool in universal algebra are ultraproducts $\prod_{i \in I} A_i / U$, where I is an infinite set indexing a system of σ -structures A_i , and U is an ultrafilter on I .

While model theory is generally considered a part of mathematical logic, universal algebra, which grew out of Alfred North Whitehead's (1898) work on abstract algebra, is part of algebra. This is reflected by their respective MSC classifications. Nevertheless model theory can be seen as an extension of universal algebra.

Finite model theory

Finite model theory is the area of model theory which has the closest ties to universal algebra. Like some parts of universal algebra, and in contrast with the other areas of model theory, it is mainly concerned with finite algebras, or more generally, with finite σ -structures for signatures σ which may contain relation symbols as in the following example:

The standard signature for graphs is $\sigma_{\text{grph}} = \{E\}$, where E is a binary relation symbol.

A graph is a σ_{grph} -structure satisfying the sentences $\forall u \forall v (uEv \rightarrow vEu)$ and $\forall u \neg(uEu)$.

A σ -homomorphism is a map that commutes with the operations and preserves the relations in σ . This definition gives rise to the usual notion of graph homomorphism, which has the interesting property that a bijective homomorphism need not be invertible. Structures are also a part of universal algebra; after all, some algebraic structures such as ordered groups have a binary relation $<$. What distinguishes finite model theory from universal

algebra is its use of more general logical sentences (as in the example above) in place of identities. (In a model-theoretic context an identity $t=t'$ is written as a sentence $\forall u_1 u_2 \dots u_n (t = t')$.)

The logics employed in finite model theory are often substantially more expressive than first-order logic, the standard logic for model theory of infinite structures.

First-order logic

Whereas universal algebra provides the semantics for a signature, logic provides the syntax. With terms, identities and quasi-identities, even universal algebra has some limited syntactic tools; first-order logic is the result of making quantification explicit and adding negation into the picture.

A first-order **formula** is built out of atomic formulas such as $R(f(x,y),z)$ or $y = x + 1$ by means of the Boolean connectives $\neg, \wedge, \vee, \rightarrow$ and prefixing of quantifiers $\forall v$ or $\exists v$. A sentence is a formula in which each occurrence of a variable is in the scope of a corresponding quantifier. Examples for formulas are φ (or $\varphi(x)$ to mark the fact that at most x is an unbound variable in φ) and ψ defined as follows:

$$\begin{aligned}\phi &= \forall u \forall v (\exists w (x \times w = u \times v) \rightarrow (\exists w (x \times w = u) \vee \exists w (x \times w = v))) \wedge x \neq 0 \wedge x \neq 1, \\ \psi &= \forall u \forall v ((u \times v = x) \rightarrow (u = x) \vee (v = x)) \wedge x \neq 0 \wedge x \neq 1.\end{aligned}$$

(Note that the equality symbol has a double meaning here.) It is intuitively clear how to translate such formulas into mathematical meaning. In the σ_{smr} -structure \mathcal{N} of the natural numbers, for example, an element n **satisfies** the formula ϕ if and only if n is a prime number. The formula ψ similarly defines irreducibility. Tarski gave a rigorous definition, sometimes called "Tarski's definition of truth", for the satisfaction relation \models , so that one easily proves:

$$\begin{aligned}\mathcal{N} \models \phi(n) &\iff n \text{ is a prime number.} \\ \mathcal{N} \models \psi(n) &\iff n \text{ is irreducible.}\end{aligned}$$

A set T of sentences is called a (first-order) theory. A theory is **satisfiable** if it has a **model** $\mathcal{M} \models T$, i.e. a structure (of the appropriate signature) which satisfies all the sentences in the set T . Consistency of a theory is usually defined in a syntactical way, but in first-order logic by the completeness theorem there is no need to distinguish between satisfiability and consistency. Therefore model theorists often use "consistent" as a synonym for "satisfiable".

A theory is called **categorical** if it determines a structure up to isomorphism, but it turns out that this definition is not useful, due to serious restrictions in the expressivity of first-order logic. The Löwenheim–Skolem theorem implies that for every theory $T^{[1]}$ which has an infinite model and for every infinite cardinal number κ , there is a model $\mathcal{M} \models T$ such that the number of elements of \mathcal{M} is exactly κ . Therefore only finite structures can be described by a categorical theory.

Lack of expressivity (when compared to higher logics such as second-order logic) has its advantages, though. For model theorists the Löwenheim–Skolem theorem is an important practical tool rather than the source of Skolem's paradox. First-order logic is in some sense (for which see Lindström's theorem) the most expressive logic for which both the Löwenheim–Skolem theorem and the compactness theorem hold:

Compactness theorem

Every unsatisfiable first-order theory has a finite unsatisfiable subset.

This important theorem, due to Gödel, is of central importance in infinite model theory, where the words "by compactness" are commonplace. One way to prove it is by means of ultraproducts. An alternative proof uses the completeness theorem, which is otherwise reduced to a marginal role in most of modern model theory.

Axiomatizability, elimination of quantifiers, and model-completeness

The first step, often trivial, for applying the methods of model theory to a class of mathematical objects such as groups, or trees in the sense of graph theory, is to choose a signature σ and represent the objects as σ -structures. The next step is to show that the class is an elementary class, i.e. axiomatizable in first-order logic (i.e. there is a theory T such that a σ -structure is in the class if and only if it satisfies T). E.g. this step fails for the trees, since connectedness cannot be expressed in first-order logic. Axiomatizability ensures that model theory can speak about the right objects. Quantifier elimination can be seen as a condition which ensures that model theory does not say too much about the objects.

A theory T has quantifier elimination if every first-order formula $\varphi(x_1, \dots, x_n)$ over its signature is equivalent modulo T to a first-order formula $\psi(x_1, \dots, x_n)$ without quantifiers, i.e. $\forall x_1 \dots \forall x_n (\varphi(x_1, \dots, x_n) \leftrightarrow \psi(x_1, \dots, x_n))$ holds in all models of T . For example the theory of algebraically closed fields in the signature $\sigma_{\text{ring}} = (\times, +, -, 0, 1)$ has quantifier elimination because every formula is equivalent to a Boolean combination of equations between polynomials.

A substructure of a σ -structure is a subset of its domain, closed under all functions in its signature σ , which is regarded as a σ -structure by restricting all functions and relations in σ to the subset. An embedding of a σ -structure \mathcal{A} into another σ -structure \mathcal{B} is a map $f: A \rightarrow B$ between the domains which can be written as an isomorphism of \mathcal{A} with a substructure of \mathcal{B} . Every embedding is an injective homomorphism, but the converse holds only if the signature contains no relation symbols.

If a theory does not have quantifier elimination, one can add additional symbols to its signature so that it does. Early model theory spent much effort on proving axiomatizability and quantifier elimination results for specific theories, especially in algebra. But often instead of quantifier elimination a weaker property suffices:

A theory T is called model-complete if every substructure of a model of T which is itself a model of T is an elementary substructure. There is a useful criterion for testing whether a substructure is an elementary substructure, called the Tarski–Vaught test. It follows from this criterion that a theory T is model-complete if and only if every first-order formula $\varphi(x_1, \dots, x_n)$ over its signature is equivalent modulo T to an existential first-order formula, i.e. a formula of the following form:

$$\exists v_1 \dots \exists v_m \psi(x_1, \dots, x_n, v_1, \dots, v_m),$$

where ψ is quantifier free. A theory that is not model-complete may or may not have a **model completion**, which is a related model-complete theory that is not, in general, an extension of the original theory. A more general notion is that of **model companions**.

Categoricity

As observed in the section on first-order logic, first-order theories cannot be categorical, i.e. they cannot describe a unique model up to isomorphism, unless that model is finite. But two famous model-theoretic theorems deal with the weaker notion of κ -categoricity for a cardinal κ . A theory T is called **κ -categorical** if any two models of T that are of cardinality κ are isomorphic. It turns out that the question of κ -categoricity depends critically on whether κ is bigger than the cardinality of the language (i.e. $\aleph_0 + |\sigma|$, where $|\sigma|$ is the cardinality of the signature). For finite or countable signatures this means that there is a fundamental difference between \aleph_0 -cardinality and κ -cardinality for uncountable κ .

The following characterization of \aleph_0 -categoricity is due independently to Ryll-Nardzewski, Engeler and Svenonius:

Ryll-Nardzewski's theorem

For a complete first-order theory T in a finite or countable signature the following conditions are equivalent:

1. T is \aleph_0 -categorical.

2. For every natural number n , the Stone space $S_n(T)$ is finite.
3. For every natural number n , the number of formulas $\varphi(x_1, \dots, x_n)$ in n free variables, up to equivalence modulo T , is finite.

\aleph_0 -categorical theories and their countable models have strong ties with oligomorphic groups. They are often constructed as Fraïssé limits.

Michael Morley's highly non-trivial result that (for countable languages) there is only *one* notion of uncountable categoricity was the starting point for modern model theory, and in particular classification theory and stability theory:

Morley's categoricity theorem

If a first-order theory T in a finite or countable signature is κ -categorical for some uncountable cardinal κ , then T is κ -categorical for all uncountable cardinals κ .

Uncountably categorical (i.e. κ -categorical for all uncountable cardinals κ) theories are from many points of view the most well-behaved theories. A theory that is both \aleph_0 -categorical and uncountably categorical is called **totally categorical**.

Model theory and set theory

Set theory (which is expressed in a countable language) has a countable model; this is known as Skolem's paradox, since there are sentences in set theory which postulate the existence of uncountable sets and yet these sentences are true in our countable model. Particularly the proof of the independence of the continuum hypothesis requires considering sets in models which appear to be uncountable when viewed from *within* the model, but are countable to someone *outside* the model.

The model-theoretic viewpoint has been useful in set theory; for example in Kurt Gödel's work on the constructible universe, which, along with the method of forcing developed by Paul Cohen can be shown to prove the (again philosophically interesting) independence of the axiom of choice and the continuum hypothesis from the other axioms of set theory.

Other basic notions of model theory

Reducts and expansions

A field or a vector space can be regarded as a (commutative) group by simply ignoring some of its structure. The corresponding notion in model theory is that of a **reduct** of a structure to a subset of the original signature. The opposite relation is called an *expansion* - e.g. the (additive) group of the rational numbers, regarded as a structure in the signature $\{+,0\}$ can be expanded to a field with the signature $\{\times,+,1,0\}$ or to an ordered group with the signature $\{+,0,<\}$.

Similarly, if σ' is a signature that extends another signature σ , then a complete σ' -theory can be restricted to σ by intersecting the set of its sentences with the set of σ -formulas. Conversely, a complete σ -theory can be regarded as a σ' -theory, and one can extend it (in more than one way) to a complete σ' -theory. The terms reduct and expansion are sometimes applied to this relation as well.

Interpretability

Given a mathematical structure, there are very often associated structures which can be constructed as a quotient of part of the original structure via an equivalence relation. An important example is a quotient group of a group.

One might say that to understand the full structure one must understand these quotients. When the equivalence relation is definable, we can give the previous sentence a precise meaning. We say that these structures are **interpretable**.

A key fact is that one can translate sentences from the language of the interpreted structures to the language of the original structure. Thus one can show that if a structure M interprets another whose theory is undecidable, then M itself is undecidable.

Using the compactness and completeness theorems

Gödel's completeness theorem (not to be confused with his incompleteness theorems) says that a theory has a model if and only if it is consistent, i.e. no contradiction is proved by the theory. This is the heart of model theory as it lets us answer questions about theories by looking at models and vice-versa. One should not confuse the completeness theorem with the notion of a complete theory. A complete theory is a theory that contains every sentence or its negation. Importantly, one can find a complete consistent theory extending any consistent theory. However, as shown by Gödel's incompleteness theorems only in relatively simple cases will it be possible to have a complete consistent theory that is also recursive, i.e. that can be described by a recursively enumerable set of axioms. In particular, the theory of natural numbers has no recursive complete and consistent theory. Non-recursive theories are of little practical use, since it is undecidable if a proposed axiom is indeed an axiom, making proof-checking a supertask.

The compactness theorem states that a set of sentences S is satisfiable if every finite subset of S is satisfiable. In the context of proof theory the analogous statement is trivial, since every proof can have only a finite number of antecedents used in the proof. In the context of model theory, however, this proof is somewhat more difficult. There are two well known proofs, one by Gödel (which goes via proofs) and one by Malcev (which is more direct and allows us to restrict the cardinality of the resulting model).

Model theory is usually concerned with first-order logic, and many important results (such as the completeness and compactness theorems) fail in second-order logic or other alternatives. In first-order logic all infinite cardinals look the same to a language which is countable. This is expressed in the Löwenheim–Skolem theorems, which state that any countable theory with an infinite model \mathfrak{Q} has models of all infinite cardinalities (at least that of the language) which agree with \mathfrak{Q} on all sentences, i.e. they are 'elementarily equivalent'.

Types

Fix an L -structure M , and a natural number n . The set of definable subsets of M^n over some parameters A is a Boolean algebra. By Stone's representation theorem for Boolean algebras there is a natural dual notion to this. One can consider this to be the topological space consisting of maximal consistent sets of formulae over A . We call this the space of (complete) n -types over A , and write $S_n(A)$.

Now consider an element $m \in M^n$. Then the set of all formulae ϕ with parameters in A in free variables x_1, \dots, x_n so that $M \models \phi(m)$ is consistent and maximal such. It is called the *type* of m over A .

One can show that for any n -type p , there exists some elementary extension N of M and some $a \in N^n$ so that p is the type of a over A .

Many important properties in model theory can be expressed with types. Further many proofs go via constructing models with elements that contain elements with certain types and then using these elements.

Illustrative Example: Suppose M is an algebraically closed field. The theory has quantifier elimination. This allows us to show that a type is determined exactly by the polynomial equations it contains. Thus the space of n

-types over a subfield A is bijective with the set of prime ideals of the polynomial ring $A[x_1, \dots, x_n]$. This is the same set as the spectrum of $A[x_1, \dots, x_n]$. Note however that the topology considered on the type space is the constructible topology: a set of types is basic open iff it is of the form $\{p : f(x) = 0 \in p\}$ or of the form $\{p : f(x) \neq 0 \in p\}$. This is finer than the Zariski topology.

Early history

Model theory as a subject has existed since approximately the middle of the 20th century. However some earlier research, especially in mathematical logic, is often regarded as being of a model-theoretical nature in retrospect. The first significant result in what is now model theory was a special case of the downward Löwenheim–Skolem theorem, published by Leopold Löwenheim in 1915. The compactness theorem was implicit in work by Thoralf Skolem,^[2] but it was first published in 1930, as a lemma in Kurt Gödel's proof of his completeness theorem. The Löwenheim–Skolem's theorem and the compactness theorem received their respective general forms in 1936 and 1941 from Anatoly Maltsev.

See also

- Axiomatizable class
- Compactness theorem
- Descriptive complexity
- Elementary equivalence
- First-order theories
- Forcing
- Hyperreal number
- Institutional model theory
- Kripke semantics
- Löwenheim–Skolem theorem
- Proof theory
- Saturated model

References

Canonical textbooks

- Chang, Chen Chung; Keisler, H. Jerome (1990) [1973]. *Model Theory*. Studies in Logic and the Foundations of Mathematics (3rd ed.). Elsevier. ISBN 978-0-444-88054-3
- Hodges, Wilfrid (1997). *A shorter model theory*. Cambridge: Cambridge University Press. ISBN 978-0-521-58713-6

Other textbooks

- Bell, John L.; Slomson, Alan B. (2006) [1969]. *Models and Ultraproducts: An Introduction* (reprint of 1974 ed.). Dover Publications. ISBN 0-486-44979-3.
- Ebbinghaus, Heinz-Dieter; Flum, Jörg; Thomas, Wolfgang (1994). *Mathematical Logic*. Springer. ISBN 0-38794258-0.
- Hinman, Peter G. (2005). *Fundamentals of Mathematical Logic*. A K Peters. ISBN 1-568-81262-0.
- Hodges, Wilfrid (1993). *Model theory*. Cambridge University Press. ISBN 0-521-30442-3.
- Marker, David (2002). *Model Theory: An Introduction*. Graduate Texts in Mathematics 217. Springer. ISBN 0-387-98760-6.
- Poizat, Bruno (2000). *A Course in Model Theory*. Springer. ISBN 0-387-98655-3.
- Rothmaler, Philipp (2000). *Introduction to Model Theory* (new ed.). Taylor & Francis. ISBN 9056993135.

Free online texts

- Chatzidakis, Zoe (2001). *Introduction to Model Theory* ^[3]. pp. 26 pages in DVI format.
- Pillay, Anand (2002). *Lecture Notes – Model Theory* ^[4]. pp. 61 pages.
- Hodges, Wilfrid, *First-order Model theory* ^[22]. The Stanford Encyclopedia Of Philosophy, E. Zalta (ed.).
- Simmons, Harold (2004), *An introduction to Good old fashioned model theory* ^[5]. Notes of an introductory course for postgraduates (with exercises).
- J. Barwise and S. Feferman (editors), *Model-Theoretic Logics* ^[6], Perspectives in Mathematical Logic, Volume 8 New York: Springer-Verlag, 1985.

References

- [1] In a countable signature. The theorem has a straightforward generalization to uncountable signatures.
- [2] *All three commentators [i.e. Vaught, van Heijenoort and Dreben] agree that both the completeness and compactness theorems were implicit in Skolem 1923 [...]*, Dawson (1993).
- [3] <http://www.logique.jussieu.fr/~zoe/papiers/MTluminy.dvi>
- [4] http://www.math.uiuc.edu/People/pillay/lecturenotes_modeltheory.pdf
- [5] <http://www.cs.man.ac.uk/~hsimmons/BOOKS/ModelTheory.pdf>
- [6] <http://projecteuclid.org/euclid.pl/1235417263>

Proof theory

Proof theory is a branch of mathematical logic that represents proofs as formal mathematical objects, facilitating their analysis by mathematical techniques. Proofs are typically presented as inductively-defined data structures such as plain lists, boxed lists, or trees, which are constructed according to the axioms and rules of inference of the logical system. As such, proof theory is syntactic in nature, in contrast to model theory, which is semantic in nature. Together with model theory, axiomatic set theory, and recursion theory, proof theory is one of the so-called *four pillars* of the foundations of mathematics.^[1]

Proof theory is important in philosophical logic, where the primary interest is in the idea of a proof-theoretic semantics, an idea which depends upon technical ideas in structural proof theory to be feasible.

History

Although the formalisation of logic was much advanced by the work of such figures as Gottlob Frege, Giuseppe Peano, Bertrand Russell, and Richard Dedekind, the story of modern proof theory is often seen as being established by David Hilbert, who initiated what is called Hilbert's program in the foundations of mathematics. Kurt Gödel's seminal work on proof theory first advanced, then refuted this program: his completeness theorem initially seemed to bode well for Hilbert's aim of reducing all mathematics to a finitist formal system; then his incompleteness theorems showed that this is unattainable. All of this work was carried out with the proof calculi called the Hilbert systems.

In parallel, the foundations of structural proof theory were being founded. Jan Łukasiewicz suggested in 1926 that one could improve on Hilbert systems as a basis for the axiomatic presentation of logic if one allowed the drawing of conclusions from assumptions in the inference rules of the logic. In response to this Stanisław Jaśkowski (1929) and Gerhard Gentzen (1934) independently provided such systems, called calculi of natural deduction, with Gentzen's approach introducing the idea of symmetry between the grounds for asserting propositions, expressed in introduction rules, and the consequences of accepting propositions in the elimination rules, an idea that has proved very important in proof theory^[2]. Gentzen (1934) further introduced the idea of the sequent calculus, a calculus advanced in a similar spirit that better expressed the duality of the logical connectives^[3], and went on to make fundamental advances in the formalisation of intuitionistic logic, and provide the first combinatorial proof of the consistency of Peano arithmetic. Together, the presentation of natural deduction and the sequent calculus introduced the

fundamental idea of analytic proof to proof theory,

Formal and informal proof

The *informal* proofs of everyday mathematical practice are unlike the *formal* proofs of proof theory. They are rather like high-level sketches that would allow an expert to reconstruct a formal proof at least in principle, given enough time and patience. For most mathematicians, writing a fully formal proof is too pedantic and long-winded to be in common use.

Formal proofs are constructed with the help of computers in interactive theorem proving. Significantly, these proofs can be checked automatically, also by computer. (Checking formal proofs is usually simple, whereas *finding* proofs (automated theorem proving) is generally hard.) An informal proof in the mathematics literature, by contrast, requires weeks of peer review to be checked, and may still contain errors.

Kinds of proof calculi

The three most well-known styles of proof calculi are:

- The Hilbert calculi
- The natural deduction calculi
- The sequent calculi

Each of these can give a complete and axiomatic formalization of propositional or predicate logic of either the classical or intuitionistic flavour, almost any modal logic, and many substructural logics, such as relevance logic or linear logic. Indeed it is unusual to find a logic that resists being represented in one of these calculi.

Consistency proofs

As previously mentioned, the spur for the mathematical investigation of proofs in formal theories was Hilbert's program. The central idea of this program was that if we could give finitary proofs of consistency for all the sophisticated formal theories needed by mathematicians, then we could ground these theories by means of a metamathematical argument, which shows that all of their purely universal assertions (more technically their provable Π_1^0 sentences) are finitarily true; once so grounded we do not care about the non-finitary meaning of their existential theorems, regarding these as pseudo-meaningful stipulations of the existence of ideal entities.

The failure of the program was induced by Kurt Gödel's incompleteness theorems, which showed that any ω -consistent theory that is sufficiently strong to express certain simple arithmetic truths, cannot prove its own consistency, which on Gödel's formulation is a Π_1^0 sentence.

Much investigation has been carried out on this topic since, which has in particular led to:

- Refinement of Gödel's result, particularly J. Barkley Rosser's refinement, weakening the above requirement of ω -consistency to simple consistency;
- Axiomatisation of the core of Gödel's result in terms of a modal language, provability logic;
- Transfinite iteration of theories, due to Alan Turing and Solomon Feferman;
- The recent discovery of self-verifying theories, systems strong enough to talk about themselves, but too weak to carry out the diagonal argument that is the key to Gödel's unprovability argument.

See also Mathematical logic

Structural proof theory

Structural proof theory is the subdiscipline of proof theory that studies proof calculi that support a notion of analytic proof. The notion of analytic proof was introduced by Gentzen for the sequent calculus; there the analytic proofs are those that are cut-free. His natural deduction calculus also supports a notion of analytic proof, as shown by Dag Prawitz. The definition is slightly more complex: we say the analytic proofs are the normal forms, which are related to the notion of normal form in term rewriting. More exotic proof calculi such as Jean-Yves Girard's proof nets also support a notion of analytic proof.

Structural proof theory is connected to type theory by means of the Curry-Howard correspondence, which observes a structural analogy between the process of normalisation in the natural deduction calculus and beta reduction in the typed lambda calculus. This provides the foundation for the intuitionistic type theory developed by Per Martin-Löf, and is often extended to a three way correspondence, the third leg of which are the cartesian closed categories.

Proof-theoretic semantics

In linguistics, type-logical grammar, categorial grammar and Montague grammar apply formalisms based on structural proof theory to give a formal natural language semantics.

Tableau systems

Analytic tableaux apply the central idea of analytic proof from structural proof theory to provide decision procedures and semi-decision procedures for a wide range of logics.

Ordinal analysis

Ordinal analysis is a powerful technique for providing combinatorial consistency proofs for theories formalising arithmetic and analysis.

Logics from proof analysis

Several important logics have come from insights into logical structure arising in structural proof theory.

See also

- Proof techniques
- Intermediate logics

References

- J. Avigad, E.H. Reck (2001). "Clarifying the nature of the infinite": the development of metamathematics and proof theory ^[4]. Carnegie-Mellon Technical Report CMU-PHIL-120.
- J. Barwise (ed., 1978). Handbook of Mathematical Logic. North-Holland.
- 2πix.com: Logic ^[5] Part of a series of articles covering mathematics and logic.
- A. S. Troelstra, H. Schwichtenberg (1996). *Basic Proof Theory*. In series *Cambridge Tracts in Theoretical Computer Science*, Cambridge University Press, ISBN 0-521-77911-1.
- G. Gentzen (1935/1969). Investigations into logical deduction. In M. E. Szabo, editor, *Collected Papers of Gerhard Gentzen*. North-Holland. Translated by Szabo from "Untersuchungen über das logische Schliessen", *Mathematisches Zeitschrift* 39: 176-210, 405-431.
- L. Moreno-Armella & B.Sriraman (2005). *Structural Stability and Dynamic Geometry: Some Ideas on Situated Proof*. *International Reviews on Mathematical Education*. Vol. 37, no.3, pp. 130–139 [6]

- J. von Plato (2008). The Development of Proof Theory ^[7]. Stanford Encyclopedia of Philosophy.
- Wang, Hao (1981). *Popular Lectures on Mathematical Logic*. Van Nostrand Reinhold Company. ISBN 0442231091.

References

- [1] E.g., Wang (1981), pp. 3–4, and Barwise (1978).
- [2] Prawitz (1965).
- [3] Girard, Lafont, and Taylor (1988).
- [4] <http://www.andrew.cmu.edu/user/avigad/Papers/infinite.pdf>
- [5] <http://2piix.com/articles/title/Logic/>
- [6] <http://www.springerlink.com/content/n602313107541846/?p=74ab8879ce75445da488d5744cbc3818&pi=0>
- [7] <http://plato.stanford.edu/entries/proof-theory-development/>

Sequent calculus

In proof theory and mathematical logic, **sequent calculus** is a family of formal systems sharing a certain style of inference and certain formal properties. The first sequent calculi, systems **LK** and **LJ**, were introduced by Gerhard Gentzen in 1934 as a tool for studying natural deduction in first-order logic (in classical and intuitionistic versions, respectively). Gentzen's so-called "Main Theorem" (*Hauptsatz*) about LK and LJ was the cut-elimination theorem, a result with far-reaching meta-theoretic consequences, including consistency. Gentzen further demonstrated the power and flexibility of this technique a few years later, applying a cut-elimination argument to give a (transfinite) proof of the consistency of Peano arithmetic, in surprising response to Gödel's incompleteness theorems. Since this early work, sequent calculi (also called **Gentzen systems**) and the general concepts relating to them have been widely applied in the fields of proof theory, mathematical logic, and automated deduction.

Introduction

One way to classify different styles of deduction systems is to look at the form of *judgments* in the system, *i.e.*, which things may appear as the conclusion of a (sub)proof. The simplest judgment form is used in Hilbert-style deduction systems, where a judgment has the form

$$B$$

where B is any formula of first-order-logic (or whatever logic the deduction system applies to, *e.g.*, propositional calculus or a higher-order logic or a modal logic). The theorems are those formulae that appear as the concluding judgment in a valid proof. A Hilbert-style system needs no distinction between formulae and judgments; we make one here solely for comparison with the cases that follow.

The price paid for the simple syntax of a Hilbert-style system is that complete formal proofs tend to get extremely long. Concrete arguments about proofs in such a system almost always appeal to the deduction theorem. This leads to the idea of including the deduction theorem as a formal rule in the system, which happens in natural deduction. In natural deduction, judgments have the shape

$$A_1, A_2, \dots, A_n \vdash B$$

where the A_i 's and B are again formulae and $n \geq 0$. In words, a judgment consists of a list (possibly empty) of formulae on the left-hand side of a turnstile symbol " \vdash ", with a single formula on the right-hand side. The theorems are those formulae B such that $\vdash B$ (with an empty left-hand side) is the conclusion of a valid proof. (In some presentations of natural deduction, the A_i 's and the turnstile are not written down explicitly; instead a two-dimensional notation from which they can be inferred is used).

The standard semantics of a judgment in natural deduction is that it asserts that whenever^[1] A_1, A_2 , etc., are all true, B will also be true. The judgments

$$A_1, \dots, A_n \vdash B \quad \text{and} \quad \vdash (A_1 \wedge \dots \wedge A_n) \rightarrow B$$

are equivalent in the strong sense that a proof of either one may be extended to a proof of the other.

Finally, *sequent calculus* generalizes the form of a natural deduction judgment to

$$A_1, \dots, A_n \vdash B_1, \dots, B_k,$$

a syntactic object called a **sequent**. Again, A_i and B_i are formulae, and n and k are nonnegative integers, that is, the left-hand-side or the right-hand-side (or neither or both) may be empty. As in natural deduction, theorems are those B where $\vdash B$ is the conclusion of a valid proof.

The standard semantics of a sequent is an assertion that whenever *all* of the A_i 's is true, *at least one* of the B_i will also be true. One way to express this is that a comma to the left of the turnstile should be thought of as an "and", and a comma to the right of the turnstile should be thought of as an (inclusive) "or". The sequents

$$A_1, \dots, A_n \vdash B_1, \dots, B_k \quad \text{and} \quad \vdash (A_1 \wedge \dots \wedge A_n) \rightarrow (B_1 \vee \dots \vee B_k)$$

are equivalent in the strong sense that a proof of either one may be extended to a proof of the other.

At first sight, this extension of the judgment form may appear to be a strange complication — it is not motivated by an obvious shortcoming of natural deduction, and it is initially confusing that the comma seems to mean entirely different things on the two sides of the turnstile. However, in a classical context the semantics of the sequent can also (by propositional tautology) be expressed either as

$$\neg A_1 \vee \neg A_2 \vee \dots \vee \neg A_n \vee B_1 \vee B_2 \vee \dots \vee B_k$$

(at least one of the A's is false, or one of the B's is true) or as

$$\neg(A_1 \wedge A_2 \wedge \dots \wedge A_n \wedge \neg B_1 \wedge \neg B_2 \wedge \dots \wedge \neg B_k)$$

(it cannot be the case that all of the A's are true and all of the B's are false). In these formulations, the only difference between formulae on either side of the turnstile is that one side is negated. Thus, swapping left for right in a sequent corresponds to negating all of the constituent formulae. This means that a symmetry such as De Morgan's laws, which manifests itself as logical negation on the semantic level, translates directly into a left-right symmetry of sequents — and indeed, the inference rules in sequent calculus for dealing with conjunction (\wedge) are mirror images of those dealing with disjunction (\vee).

Many logicians feel that this symmetric presentation offers a deeper insight in the structure of the logic than other styles of proof system, where the classical duality of negation is not as apparent in the rules.

The system LK

This section introduces the rules of the sequent calculus **LK** (which is short for “**l**ogistischer **k**lassischer **K**alkül”), as introduced by Gentzen in 1934.^[2] A (formal) proof in this calculus is a sequence of sequents, where each of the sequents is derivable from sequents appearing earlier in the sequence by using one of the rules below.

Inference rules

The following notation will be used:

- \vdash known as the turnstile, separates the *assumptions* on the left from the *propositions* on the right
- A and B denote formulae of first-order predicate logic (one may also restrict this to propositional logic),
- Γ, Δ, Σ , and Π are finite (possibly empty) sequences of formulae, called contexts,
 - when on the *left* of the \vdash , the sequence of formulas is considered *conjunctively* (all assumed to hold at the same time),

- while on the *right* of the \vdash , the sequence of formulas is considered *disjunctively* (at least one of the formulas must hold for any assignment of variables),
- t denotes an arbitrary term,
- x and y denote variables,
- $A[t/x]$ denotes the formula that is obtained by substituting the term t for the variable x in formula A ,
- a variable is said to occur free within a formula if it occurs outside the scope of quantifiers \forall or \exists .
- WL and WR stand for *Weakening Left/Right*, CL and CR for *Contraction*, and PL and PR for *Permutation*.

| | |
|------------------------|-------------------------|
| Axiom: | Cut: |
| | |
| Left logical rules: | Right logical rules: |
| | |
| Left structural rules: | Right structural rules: |
| | |

Restrictions: In the rules $(\exists R)$ and $(\exists L)$, the variable y must not be free within Γ , A , or Δ .

An intuitive explanation

The above rules can be divided into two major groups: *logical* and *structural* ones. Each of the logical rules introduces a new logical formula either on the left or on the right of the turnstile \vdash . In contrast, the structural rules operate on the structure of the sequents, ignoring the exact shape of the formulae. The two exceptions to this general scheme are the axiom of identity (I) and the rule of (Cut).

Although stated in a formal way, the above rules allow for a very intuitive reading in terms of classical logic. Consider, for example, the rule $(\wedge L_1)$. It says that, whenever one can prove that Δ can be concluded from some sequence of formulae that contain A , then one can also conclude Δ from the (stronger) assumption, that $A \wedge B$ holds. Likewise, the rule $(\neg R)$ states that, if Γ and A suffice to conclude Δ , then from Γ alone one can either still conclude Δ or A must be false, i.e. $\neg A$ holds. All the rules can be interpreted in this way.

For an intuition about the quantifier rules, consider the rule $(\forall R)$. Of course concluding that $\forall x A$ holds just from the fact that $A[y/x]$ is true is not in general possible. If, however, the variable y is not mentioned elsewhere (i.e. it can still be chosen freely, without influencing the other formulae), then one may assume, that $A[y/x]$ holds for any value of y . The other rules should then be pretty straightforward.

Instead of viewing the rules as descriptions for legal derivations in predicate logic, one may also consider them as instructions for the construction of a proof for a given statement. In this case the rules can be read bottom-up; for example, $(\wedge R)$ says that, to prove that $A \wedge B$ follows from the assumptions Γ and Σ , it suffices to prove that A can be concluded from Γ and B can be concluded from Σ , respectively. Note that, given some antecedent, it is not clear how

this is to be split into Γ and Σ . However, there are only finitely many possibilities to be checked since the antecedent by assumption is finite. This also illustrates how proof theory can be viewed as operating on proofs in a combinatorial fashion: given proofs for both A and B , one can construct a proof for $A \wedge B$.

When looking for some proof, most of the rules offer more or less direct recipes of how to do this. The rule of cut is different: It states that, when a formula A can be concluded and this formula may also serve as a premise for concluding other statements, then the formula A can be "cut out" and the respective derivations are joined. When constructing a proof bottom-up, this creates the problem of guessing A (since it does not appear at all below). The cut-elimination theorem is thus crucial to the applications of sequent calculus in automated deduction: it states that all uses of the cut rule can be eliminated from a proof, implying that any provable sequent can be given a *cut-free* proof.

The second rule that is somewhat special is the axiom of identity (I). The intuitive reading of this is obvious: every formula proves itself. Like the cut rule, the axiom of identity is somewhat redundant: the completeness of atomic initial sequents states that the rule can be restricted to atomic formulas without any loss of provability.

Observe that all rules have mirror companions, except the ones for implication. This reflects the fact that the usual language of first-order logic does not include the "is not implied by" connective \nrightarrow that would be the De Morgan dual of implication. Adding such a connective with its natural rules would make the calculus completely left-right symmetric.

An example derivation

As for an example, this is the sequential derivation of $\vdash (A \vee \neg A)$, known as the *Law of excluded middle* (*tertium non datur* in Latin).

$$\begin{array}{c}
 \frac{}{A \vdash A} \text{ (I)} \\
 \frac{}{\vdash \neg A, A} (\neg R) \\
 \frac{}{\vdash A \vee \neg A, A} (\vee R_2) \\
 \frac{}{\vdash A, A \vee \neg A} (PR) \\
 \frac{}{\vdash A \vee \neg A, A \vee \neg A} (\vee R_1) \\
 \frac{}{\vdash A \vee \neg A} (CR)
 \end{array}$$

This derivation also emphasizes the strictly formal structure of the sequent calculus. For example, the right logical rules as defined above always act on the first formula of the right sequent, such that the application of (PR) is formally required. Note, though, that this is in part an artifact of the presentation, in the original style of Gentzen. A common simplification is the use of multisets of formulas, rather than sequences, eliminating the need for an explicit permutation rule.

Structural rules

The structural rules deserve some additional discussion.

Weakening (W) allows the addition of arbitrary elements to a sequence. Intuitively, this is allowed in the antecedent because we can always add assumptions to our proof, and in the succedent because we can always allow for alternative conclusions.

Contraction (C) and Permutation (P) assure that neither the order (P) nor the multiplicity of occurrences (C) of elements of the sequences matters. Thus, one could instead of sequences also consider sets.

The extra effort of using sequences, however, is justified since part or all of the structural rules may be omitted. Doing so, one obtains the so-called substructural logics.

Properties of the system LK

This system of rules can be shown to be both sound and complete with respect to first-order logic, i.e. a statement A follows semantically from a set of premises Γ ($\Gamma \models A$) iff the sequent $\Gamma \vdash A$ can be derived by the above rules.

In the sequent calculus, the rule of cut is admissible. This result is also referred to as Gentzen's *Hauptsatz* ("Main Theorem").

Variants

The above rules can be modified in various ways:

Minor structural alternatives

There is some freedom of choice regarding the technical details of how sequents and structural rules are formalized. As long as every derivation in LK can be effectively transformed to a derivation using the new rules and vice versa, the modified rules may still be called LK.

First of all, as mentioned above, the sequents can be viewed to consist of sets or multisets. In this case, the rules for permuting and (when using sets) contracting formulae are obsolete.

The rule of weakening will become admissible, when the axiom (I) is changed, such that any sequent of the form $\Gamma, A \vdash A, \Delta$ can be concluded. This means that A proves A in any context. Any weakening that appears in a derivation can then be performed right at the start. This may be a convenient change when constructing proofs bottom-up.

Independent of these one may also change the way in which contexts are split within the rules: In the cases ($\wedge R$), ($\vee L$), and ($\rightarrow L$) the left context is somehow split into Γ and Σ when going upwards. Since contraction allows for the duplication of these, one may assume that the full context is used in both branches of the derivation. By doing this, one assures that no important premises are lost in the wrong branch. Using weakening, the irrelevant parts of the context can be eliminated later.

Substructural logics

Alternatively, one may restrict or forbid the use of some of the structural rules. This yields a variety of substructural logic systems. They are generally weaker than LK (*i.e.*, they have fewer theorems), and thus not complete with respect to the standard semantics of first-order logic. However, they have other interesting properties that have led to applications in theoretical computer science and artificial intelligence.

Intuitionistic sequent calculus: System LJ

Surprisingly, some small changes in the rules of LK suffice to turn it into a proof system for intuitionistic logic. To this end, one has to restrict to sequents with exactly one formula on the right-hand side, and modify the rules to maintain this invariant. For example, ($\vee L$) is reformulated as follows (where C is an arbitrary formula):

$$\frac{\Gamma, A \vdash C \quad \Sigma, B \vdash C}{\Gamma, \Sigma, A \vee B \vdash C} \quad (\vee L)$$

The resulting system is called LJ. It is sound and complete with respect to intuitionistic logic and admits a similar cut-elimination proof.

See also

- Resolution (logic)

References

- Girard, Jean-Yves; Paul Taylor, Yves Lafont (1990) [1989]. *Proofs and Types* ^[3]. Cambridge University Press (Cambridge Tracts in Theoretical Computer Science, 7). ISBN 0-521-37181-3.

External links

- A Brief Diversion: Sequent Calculus ^[4]

References

- [1] Here, "whenever" is used as an informal abbreviation "for every assignment of values to the free variables in the judgment"
- [2] Gentzen, Gerhard (1934/1935). "Untersuchungen über das logische Schließen. I". *Mathematische Zeitschrift* **39** (2): 176–210. doi:10.1007/BF01201353.
- [3] <http://www.paultaylor.eu/stable/Proofs%2BTypes.html>
- [4] http://scienceblogs.com/goodmath/2006/07/a_brief_diversion_sequent_calc.php

Python (programming language)

| | |
|---|--|
|  | |
| Paradigm | multi-paradigm: object-oriented, imperative, functional, reflective |
| Appeared in | 1991 |
| Designed by | Guido van Rossum |
| Developer | Python Software Foundation |
| Stable release | 3.1.2/ March 21, 2010 2.7/ July 3, 2010 |
| Typing discipline | duck, dynamic, strong |
| Major implementations | CPython, IronPython, Jython, Python for S60, PyPy, Unladen Swallow |
| Dialects | Stackless Python, RPython |
| Influenced by | ABC, ALGOL 68, ^[1] C, Haskell, Icon, Lisp, Modula-3, Perl, Java |
| Influenced | Boo, Cobra, D, Falcon, Groovy, Ruby, JavaScript |
| OS | Cross-platform |
| License | Python Software Foundation License |
| Usual file extensions | .py, .pyw, .pyc, .pyo, .pyd |
| Website | www.python.org ^[2] |
|  Python Programming at Wikibooks | |

Python is a general-purpose high-level programming language^[3] whose design philosophy emphasizes code readability.^[4] Python aims to combine "remarkable power with very clear syntax",^[5] and its standard library is large and comprehensive. Its use of indentation for block delimiters is unusual among popular programming languages.

Python supports multiple programming paradigms, primarily but not limited to object oriented, imperative and, to a lesser extent, functional programming styles. It features a fully dynamic type system and automatic memory management, similar to that of Scheme, Ruby, Perl, and Tcl. Like other dynamic languages, Python is often used as a scripting language, but is also used in a wide range of non-scripting contexts.

The reference implementation of Python (CPython) is free and open source software and has a community-based development model, as do all or nearly all of its alternative implementations. CPython is managed by the non-profit Python Software Foundation.

History

Python was conceived in the late 1980s^[6] and its implementation was started in December 1989^[7] by Guido van Rossum at CWI in the Netherlands as a successor to the ABC programming language (itself inspired by SETL)^[8] capable of exception handling and interfacing with the Amoeba operating system.^[9] Van Rossum is Python's principal author, and his continuing central role in deciding the direction of Python is reflected in the title given to him by the Python community, *Benevolent Dictator for Life* (BDFL).

Python 2.0 was released on 16 October 2000, with many major new features including a full garbage collector and support for Unicode. However, the most important change was to the development process itself, with a shift to a more transparent and community-backed process.^[10] Python 3.0, a major, backwards-incompatible release, was released on 3 December 2008^[11] after a long period of testing. Many of its major features have been backported to the backwards-compatible Python 2.6.^[12]

Programming philosophy

Python is a multi-paradigm programming language. Rather than forcing programmers to adopt a particular style of programming, it permits several styles: object-oriented programming and structured programming are fully supported, and there are a number of language features which support functional programming and aspect-oriented programming (including by metaprogramming^[13] and by magic methods).^[14] Many other paradigms are supported using extensions, such as pyDBC^[15] and Contracts for Python^[16] which allow Design by Contract.

Python uses dynamic typing and a combination of reference counting and a cycle-detecting garbage collector for memory management. An important feature of Python is dynamic name resolution (late binding), which binds method and variable names during program execution.

Rather than requiring all desired functionality to be built into the language's core, Python was designed to be highly extensible. New built-in modules can be easily written in C, C++ or Cython. Python can also be used as an extension language for existing modules and applications that need a programmable interface. This design of a small core language with a large standard library and an easily extensible interpreter was intended by Van Rossum from the very start because of his frustrations with ABC (which espoused the opposite mindset).^[6]

The design of Python offers only limited support for functional programming in the Lisp tradition. However, Python's design philosophy exhibits significant similarities to those of minimalist Lisp-family languages, such as Scheme. The library has two modules (itertools and functools) that implement proven functional tools borrowed from Haskell and Standard ML.^[17]

While offering choice in coding methodology, the Python philosophy rejects exuberant syntax, such as in Perl, in favor of a sparser, less-cluttered grammar. Python's developers expressly promote a particular "culture" or ideology based on what they want the language to be, favoring language forms they see as "beautiful", "explicit" and "simple". As Alex Martelli put it in his *Python Cookbook* (2nd ed., p. 230): "To describe something as clever is NOT considered a compliment in the Python culture." Python's philosophy rejects the Perl "there is more than one way to do it" approach to language design in favor of "there should be one—and preferably only one—obvious way to do it".^[18]

Python's developers eschew premature optimization, and moreover, reject patches to non-critical parts of CPython which would offer a marginal increase in speed at the cost of clarity.^[19] Python is sometimes described as "slow".^[20] However, by the Pareto principle, most problems and sections of programs are not speed critical. When speed is a problem, Python programmers tend to try to optimize bottlenecks by algorithm improvements or data structure changes, using a JIT compiler such as Pyco, rewriting the time-critical functions in "closer to the metal" languages such as C, or by translating (a dialect of) Python code to C code using tools like Cython.^[21]

The core philosophy of the language is summarized by the document "PEP 20 (The Zen of Python)".^[18]

Name and neologisms

An important goal of the Python developers is making Python fun to use. This is reflected in the origin of the name (based on the television series *Monty Python's Flying Circus*), in the common practice of using Monty Python references in example code, and in an occasionally playful approach to tutorials and reference materials.^[22] ^[23] For example, the metasyntactic variables often used in Python literature are *spam* and *eggs*, instead of the traditional *foo* and *bar*.

A common neologism in the Python community is *pythonic*, which can have a wide range of meanings related to program style. To say that a piece of code is pythonic is to say that it uses Python idioms well, that it is natural or shows fluency in the language. Likewise, to say of an interface or language feature that it is pythonic is to say that it works well with Python idioms, that its use meshes well with the rest of the language.

In contrast, a mark of *unpythonic* code is that it attempts to write C++ (or Lisp, Perl, or Java) code in Python—that is, provides a rough transcription rather than an idiomatic translation of forms from another language. The concept of pythonicity is tightly bound to Python's minimalist philosophy of readability and avoiding the "there's more than one way to do it" approach. Unreadable code or incomprehensible idioms are unpythonic.

Users and admirers of Python—most especially those considered knowledgeable or experienced—are often referred to as *Pythonists*, *Pythonistas*, and *Pythoneers*.^[24]

The prefix *Py* can be used to show that something is related to Python. Examples of the use of this prefix in names of Python applications or libraries include Pygame, a binding of SDL to Python (commonly used to create games); PyS60, an implementation for the Symbian Series 60 Operating System; PyQt and PyGTK, which bind Qt and GTK, respectively, to Python; and PyPy, a Python implementation written in Python. The prefix is also used outside of naming software packages: the major Python conference is named PyCon.

Usage

Python is often used as a scripting language for web applications, e.g. via `mod_wsgi` for the Apache web server. With Web Server Gateway Interface a standard API has been developed to facilitate these applications. Web application frameworks or application servers like Django, Pylons, TurboGears, web2py and Zope support developers in the design and maintenance of complex applications. Libraries like NumPy, SciPy and Matplotlib allow Python to be used effectively in scientific computing.

Python has been successfully embedded in a number of software products as a scripting language, including in finite element method software such as Abaqus, 3D animation packages such as Maya, MotionBuilder, Softimage, Cinema 4D, BodyPaint 3D, modo, and Blender, and 2D imaging programs like GIMP, Inkscape, Scribus, and Paint Shop Pro.^[25] ESRI is now promoting Python as the best choice for writing scripts in ArcGIS.^[26] It has even been used in several videogames.^[27]

For many operating systems, Python is a standard component; it ships with most Linux distributions, with NetBSD, and OpenBSD, and with Mac OS X and can be used from the terminal. Ubuntu uses the Ubiquity installer, while Red Hat Linux and Fedora use the Anaconda installer, and both installers are written in Python. Gentoo Linux uses Python in its package management system, Portage, and the standard tool to access it, emerge. Pardus uses it for administration and during system boot.^[28]

Python has also seen extensive use in the information security industry, including exploit development.^[29]

Among the users of Python are YouTube^[30] and the original BitTorrent client.^[31] Large organizations that make use of Python include Google,^[32] Yahoo!,^[33] CERN,^[34] NASA,^[35] and ITA.^[36] Most of the Sugar software for the One Laptop Per Child XO, now developed at Sugar Labs, is written in Python.^[37]

Syntax and semantics

Python was intended to be a highly readable language. It is designed to have an uncluttered visual layout, frequently using English keywords where other languages use punctuation. Python requires less boilerplate than traditional manifestly typed structured languages such as C or Pascal, and has a smaller number of syntactic exceptions and special cases than either of these.^[38]

Indentation

Python uses whitespace indentation, rather than curly braces or keywords, to delimit blocks (a feature also known as the off-side rule). An increase in indentation comes after certain statements; a decrease in indentation signifies the end of the current block.^[39]

```
def add5(x):
    return x+5

def dotwrite(ast):
    nodename = getNodeName()
    label=symbol.sym_name.get(int(ast[0]),ast[0])
    print '    %s [label="%s" % (nodename,label),
    if isinstance(ast[1], str):
        if ast[1].strip():
            print '= %s";' % ast[1]
        else:
            print "]"
    else:
        print "];"
        children = []
        for n, child in enumerate(ast[1:]):
            children.append(dotwrite(child))
        print '    %s -> {' % nodename,
        for name in children:
            print '%s' % name,
```

Syntax-highlighted Python 2.x code.

Statements and control flow

Python's statements include (among others):

- The if statement, which conditionally executes a block of code, along with else and elif (a contraction of else-if).
- The for statement, which iterates over an iterable object, capturing each element to a local variable for use by the attached block.
- The while statement, which executes a block of code as long as its condition is true.
- The try statement, which allows exceptions raised in its attached code block to be caught and handled by except clauses; it also ensures that clean-up code in a finally block will always be run regardless of how the block exits.
- The class statement, which executes a block of code and attaches its local namespace to a class, for use in object-oriented programming.
- The def statement, which defines a function or method.
- The with statement (from Python 2.6), which encloses a code block within a context manager (for example, acquiring a lock before the block of code is run, and releasing the lock afterwards).
- The pass statement, which serves as a NOP and can be used in place of a code block.
- The assert statement, used during debugging to check for conditions that ought to apply.
- The yield statement, which returns a value from a generator function. (From Python 2.5, yield is also an operator. This form is used to implement coroutines -- see below.)
- The print statement, which writes a value to an output stream (e.g. file or console). From Python 3, this statement is replaced by a function.
- The exec statement, which dynamically executes a string or file containing Python code in the form of one or more statements. From Python 3, this statement is replaced by a function.

Each statement has its own semantics: for example, the def statement does not execute its block immediately, unlike most other statements.

CPython does not support continuations, and according to Guido van Rossum it never will.^[40] However, better support for coroutine-like functionality is provided in 2.5, by extending Python's generators.^[41] Prior to 2.5, generators were lazy iterators; information was passed unidirectionally out of the generator. As of Python 2.5, it is possible to pass information back into a generator function.

Expressions

Python expressions are similar to languages such as C and Java. Some important notes:

- In Python 2, the `/` operator on integers does integer division, i.e. it truncates the result to an integer. In Python 3, however, the result of `/` is always a floating-point value, and a new operator `//` is introduced to do integer division. This behavior can be enabled using the statement `from __future__ import division`.
- In Python, `==` compares by value, in contrast to Java, where it compares by reference (value comparisons in Java use the *equals* method). To compare by reference in Python, use the *is* operator.
- Python uses named *and*, *or*, *not* operators rather than symbolic `&&`, `||`, `!`.
- Python has an important type of expression known as a *list comprehension*. Recent versions of Python have extended list comprehensions into a more general expression known as a *generator expression*.
- Anonymous functions are implemented using *lambda expressions*; however, these are limited in that the body can only be a single expression.
- Conditional expressions in Python are written as *y if x else z* (different in order of operands from the `?:` operator common to many other languages).
- Python makes a distinction between lists and tuples. Lists are written as `[1, 2, 3]` are mutable, and cannot be used as the keys of dictionaries (dictionary keys must be immutable in Python). Tuples are written as `(1, 2, 3)`, are immutable and thus can be used as the keys of dictionaries. The parentheses around the tuple are optional in some contexts. Tuples can appear on the left side of an equal sign; hence an expression like `x, y = y, x` can be used to swap two variables.
- Python has a "string format" operator `%`. This functions analogous to printf expressions in C, e.g. `"foo=%s bar=%d" % ("blah", 2)` evaluates to `"foo=blah bar=2"`.
- Python has various kinds of strings.
 - Either single or double quotes can be used to quote strings. Unlike in Unix shell languages, Perl or Perl-influenced languages such as Ruby or Groovy, single quotes and double quotes function identically, i.e. there is no string interpolation of `$foo` expressions.
 - There are also multi-line strings, which begin and end with a series of three single or double quotes and function like here documents in shell languages, Perl and Ruby.
 - Finally, all of the previously-mentioned string types come in "raw" varieties (denoted by placing a literal *r* before the opening quote), which do no backslash-interpolation and hence are very useful for regular expressions or Windows-style paths; compare "`@-quoting`" in C#.
- Python has slice expressions on lists, denoted as `...[left:right]` or `...[left:right:stride]`. For example, if the variable *nums* is assigned the list `[1, 3, 5, 7, 8, 13, 20]`, then:
 - `nums[2:5] == [5, 7, 8]`, i.e. the slice goes up to, but not including, the right index.
 - `nums[1:] == [3, 5, 7, 8, 13, 20]`, i.e. all elements but the first, because an omitted right index means "the end".
 - `nums[:3] == [1, 3, 5, 7]`, i.e. an omitted left index means "the start", and a negative index (either left or right) counts from the end.
 - `nums[:]` makes a copy of the list. This means `nums == nums[:]` is true but `nums is nums[:]` is false. Changes to the copy will not affect the original.
 - `nums[1:5:2] == [3, 7]`, i.e. if three numbers are given, the third is the "stride", indicating in this case that every second element will be selected.

In Python, a distinction between expressions and statements is rigidly enforced, in contrast to languages such as Common Lisp, Scheme or Ruby. This leads to some duplication of functionality, e.g.

- list comprehensions vs. "for" loops
- conditional expressions vs. "if" blocks
- The *eval()* vs. *exec()* builtins (in Python 2, *exec* is a statement declarator); *eval()* is for expressions, *exec()* is for statements.

Statements cannot be a part of an expression and so list and other comprehensions or lambda expressions, all being expressions, cannot contain statements. A particular case of this is that an assignment statement such as `a = 1` cannot form part of the conditional *expression* of a conditional statement, this has the advantage of avoiding a classic C error of mistaking an assignment token `=`, for an equality operator `==` which would remain valid C in `if (c = 1) { ... }` but `if c = 1: ...` is *invalid* Python code.

Methods

Methods on objects are functions attached to the object's class; the syntax `instance.method(argument)` is, for normal methods and functions, syntactic sugar for `Class.method(instance, argument)`. Python methods have an explicit `self` parameter to access instance data, in contrast to the implicit `self` in some other object-oriented programming languages (for example, Java, C++ or Ruby).^[42]

Typing

Python uses duck typing and has typed objects but untyped variable names. Type constraints are not checked at compile time; rather, operations on an object may fail, signifying that the given object is not of a suitable type. Despite being dynamically typed, Python is strongly typed, forbidding operations that are not well-defined (for example, adding a number to a string) rather than silently attempting to make sense of them.

Python allows programmers to define their own types using classes, which are most often used for object-oriented programming. New instances of classes are constructed by calling the class (for example, `SpamClass()` or `EggsClass()`), and the classes themselves are instances of the metaclass type (itself an instance of itself), allowing metaprogramming and reflection.

Prior to version 3.0, Python had two kinds of classes: "old-style" and "new-style". Old-style classes were eliminated in Python 3.0, making all classes new-style. In versions between 2.2 and 3.0, both kinds of classes could be used. The syntax of both styles is the same, the difference being whether the class object is inherited from, directly or indirectly (all new-style classes inherit from `object` and are instances of `type`).

Here is a summary of Python's built-in types:

| Type | Description | Syntax example |
|----------------|--|---|
| str | An immutable sequence of Unicode characters | 'Wikipedia' "Wikipedia" """Spanning multiple lines""" |
| bytes | An immutable sequence of bytes | b'Some ASCII' b"Some ASCII" |
| list | Mutable, can contain mixed types | [4.0, 'string', True] |
| tuple | Immutable, can contain mixed types | (4.0, 'string', True) |
| set, frozenset | Unordered, contains no duplicates | {4.0, 'string', True} frozenset([4.0, 'string', True]) |
| dict | A mutable group of key and value pairs | {'key1': 1.0, 3: False} |
| int | An immutable fixed precision number of unlimited magnitude | 42 |
| float | An immutable floating point number (system-defined precision) | 3.1415927 |
| complex | An immutable complex number with real number and imaginary parts | 3+2.7j |
| bool | An immutable truth value | True False |

| | | |
|------|--|-----|
| long | An immutable fixed precision number of unlimited magnitude | 10L |
|------|--|-----|

While many programming languages round the result of integer division towards zero, Python always rounds it down towards minus infinity; so that $7//3$ is 2, but $(-7)//3$ is -3 .

Python provides a round function for rounding floats to integers. Version 2.6.1 and lower use round-away-from-zero: $\text{round}(0.5)$ is 1.0, $\text{round}(-0.5)$ is -1.0. Version 3.0 and higher use round-to-even: $\text{round}(1.5)$ is 2.0, $\text{round}(2.5)$ is 2.0. The Decimal type/class in module decimal (since version 2.4) provides exact numerical representation and several rounding modes.

Implementations

CPython

The mainstream Python implementation, known as *CPython*, is written in C meeting the C89 standard.^[43] CPython compiles Python programs into intermediate bytecode,^[44] which are then executed by the virtual machine.^[45] It is distributed with a large standard library written in a mixture of C and Python. CPython ships in versions for many platforms, including Microsoft Windows and most modern Unix-like systems. CPython was intended from almost its very conception to be cross-platform; its use and development on esoteric platforms such as Amoeba, alongside more conventional ones like Unix and Mac OS, has greatly helped in this regard.^[46]

Stackless Python is a significant fork of CPython that implements microthreads; it does not use the C memory stack. It can be expected to run on approximately the same platforms that CPython runs on.

Google started a project called Unladen Swallow in 2009 with the aims of increasing the speed of the Python interpreter by 5 times and improving its multithreading ability to scale to thousands of cores.^[47]

Alternative implementations

Jython compiles the Python program into Java byte code, which can then be executed by every Java Virtual Machine implementation. This also enables the use of Java class library functions from the Python program. IronPython follows a similar approach in order to run Python programs on the .NET Common Language Runtime. PyPy is an experimental self-hosting implementation of Python, written in Python, that can output several types of bytecode, object code and intermediate languages. There also exist compilers to high-level object languages, with either unrestricted Python, a restricted subset of Python, or a language similar to Python as the source language. PyPy is of this type, compiling RPython to several languages; other examples include Pyjamas compiling to Javascript; Shed Skin compiling to C++; and Cython & Pyrex compiling to C.

In 2005 Nokia released a Python interpreter for the Series 60 mobile phones called PyS60. It includes many of the modules from the CPython implementations and some additional modules for integration with the Symbian operating system. This project has been kept up to date to run on all variants of the S60 platform and there are several third party modules available. The Nokia N900 also supports Python with gtk windows libraries, with the feature that programs can be both written and run on the device itself. There is also a Python interpreter for Windows CE devices (including Pocket PC). It is called PythonCE. There are additional tools available for easy application and GUI development.

Around 2004 and 2006, the Pyastra project and the PyMite project started porting Python to some very resource-constrained microcontrollers -- less than 4 KBytes of RAM.^[48]

ChinesePython (中蟒) is a Python programming language using Chinese language lexicon. Besides reserved words and variable names, most data type operations can be coded in Chinese as well.

Interpretational semantics

Most Python implementations (including CPython, the primary implementation) can function as a command line interpreter, for which the user enters statements sequentially and receives the results immediately. In short, Python acts as a shell. While the semantics of the other modes of execution (bytecode compilation, or compilation to native code) preserve the sequential semantics, they offer a speed boost at the cost of interactivity, so they are usually only used outside of a command-line interaction (e.g., when importing a module).

Other shells add capabilities beyond those in the basic interpreter, including IDLE and IPython. While generally following the visual style of the Python shell, they implement features like auto-completion, retention of session state, and syntax highlighting.

Some implementations can compile not only to bytecode, but can turn Python code into machine code. So far, this has only been done for restricted subsets of Python. PyPy takes this approach, naming its restricted compilable version of Python *RPython*.

Psyco is a specialising just in time compiler that integrates with CPython and transforms bytecode to machine code at runtime. The produced code is specialised for certain data types and is faster than standard Python code. Psyco is compatible with all Python code, not only a subset.^[49]

Development

Python development is conducted largely through the Python Enhancement Proposal (or "PEP") process. PEPs are standardized design documents providing general information related to Python, including proposals, descriptions, design rationales, and explanations for language features.^[50] Outstanding PEPs are reviewed and commented upon by Van Rossum, the Python project's Benevolent Dictator for Life (leader / language architect).^[51] CPython's developers also communicate over a mailing list, python-dev, which is the primary forum for discussion about the language's development; specific issues are discussed in the Roundup bug tracker maintained at python.org.^[52] Development takes place at the self-hosted `svn.python.org`.

CPython's public releases come in three types, distinguished by which part of the version number is incremented:

- backwards-incompatible versions, where code is expected to break and must be manually ported. The first part of the version number is incremented. These releases happen infrequently—for example, version 3.0 was released 8 years after 2.0.
- major or 'feature' releases, which are largely compatible but introduce new features. The second part of the version number is incremented. These releases are scheduled to occur roughly every 18 months, and each major version is supported by bugfixes for several years after its release.^[53]
- bugfix releases, which introduce no new features but fix bugs. The third and final part of the version number is incremented. These releases are made whenever a sufficient number of bugs have been fixed upstream since the last release, or roughly every 3 months. Security vulnerabilities are also patched in bugfix releases.^[54]

A number of alpha, beta, and release-candidates are also released as previews and for testing before the final release is made. Although there is a rough schedule for each release, this is often pushed back if the code is not ready. The development team monitor the state of the code by running the large unit test suite during development, and using the BuildBot continuous integration system.^[55]

Standard library

Python has a large standard library, commonly cited as one of Python's greatest strengths,^[56] providing pre-written tools suited to many tasks. This is deliberate and has been described as a "batteries included"^[57] Python philosophy. The modules of the standard library can be augmented with custom modules written in either C or Python. Recently, Boost C++ Libraries includes a library, Boost.Python, to enable interoperability between C++ and Python. Because of the wide variety of tools provided by the standard library, combined with the ability to use a lower-level language such as C and C++, which is already capable of interfacing between other libraries, Python can be a powerful glue language between languages and tools.

The standard library is particularly well tailored to writing Internet-facing applications, with a large number of standard formats and protocols (such as MIME and HTTP) already supported. Modules for creating graphical user interfaces, connecting to relational databases, arithmetic with arbitrary precision decimals, manipulating regular expressions, and doing unit testing are also included.^[58]

Some parts of the standard library are covered by specifications (for example, the WSGI implementation `wsgiref` follows PEP 333^[59]), but the majority of the modules are not. They are specified by their code, internal documentation, and test suite (if supplied). However, because most of the standard library is cross-platform Python code, there are only a few modules that must be altered or completely rewritten by alternative implementations.

The standard library is not essential to run python or embed python within an application. Blender 2.49 for instance omits most of the standard library.

Influences on other languages

Python's design and philosophy have influenced several programming languages, including:

- Pyrex and its derivative Cython are code translators that are targeted at writing fast C extensions for the CPython interpreter. The language is mostly Python with syntax extensions for C and C++ features. Both languages produce compilable C code as output.
- Boo uses indentation, a similar syntax, and a similar object model. However, Boo uses static typing and is closely integrated with the .NET framework.^[60]
- Cobra uses indentation and a similar syntax. Cobra's "Acknowledgements" document lists Python first among languages that influenced it.^[61] However, Cobra directly supports design-by-contract, unit tests and optional static typing.^[62]
- ECMAScript borrowed iterators, generators, and list comprehensions from Python.^[63]
- Go is described as incorporating the "development speed of working in a dynamic language like Python".^[64]
- Groovy was motivated by the desire to bring the Python design philosophy to Java.^[65]
- OCaml has an optional syntax, called `tw` (The Whitespace Thing), inspired by Python and Haskell.^[66]

Python's development practices have also been emulated by other languages. The practice of requiring a document describing the rationale for, and issues surrounding, a change to the language (in Python's case, a PEP) is also used in Tcl^[67] and Erlang^[68] because of Python's influence.

See also

- List of programming languages
- Comparison of computer shells
- Comparison of programming languages
- List of Python software
- List of integrated development environments for Python
- Scripting language

Further reading

- Payne, James (2010). *Beginning Python: Using Python 2.6 and Python3.1* (1st ed.). Wrox. ISBN 978-0470414637.
- Beazley, David M. (2009). *Python Essential Reference* (4th ed.). Addison-Wesley Professional. ISBN 978-0672329784.
- Summerfield, Mark (2009). *Programming in Python 3* ^[69] (2nd ed.). Addison-Wesley Professional. ISBN 978-0321680563.
- Lutz, Mark (2009). *Learning Python* (4th ed.). O'Reilly Media. ISBN 978-0596158064.
- Hamilton, Naomi (5 August 2008). "The A-Z of Programming Languages: Python" ^[70]. *Computerworld*. Retrieved 2010-03-31. - An interview with Guido Van Rossum on Python
- Martelli, Alex; Ravenscroft, Anna; Ascher, David (2005). *Python Cookbook* (2nd ed.). O'Reilly Media. ISBN 978-0596007973.
- Pilgrim, Mark (2004). *Dive Into Python* ^[71]. Apress. ISBN 978-1590593561.
- Downey, Allen B.. *Think Python: How to Think Like a Computer Scientist* ^[72].

External links

- Official Python website ^[2]
- Python ^[73] at the Open Directory Project


References

- [1] "Interview with Guido van Rossum" (<http://www.amk.ca/python/writing/gvr-interview>). July 1998. . Retrieved 29 2007.
- [2] <http://www.python.org/>
- [3] "What is Python Good For?" (<http://docs.python.org/faq/general.html#what-is-python-good-for>). *General Python FAQ*. Python Foundation. . Retrieved 2008-09-05.
- [4] "What is Python? Executive Summary" (<http://www.python.org/doc/essays/blurbl/>). *Python documentation*. Python Foundation. . Retrieved 2007-03-21.
- [5] "General Python FAQ" (<http://www.python.org/doc/faq/general/#what-is-python>). *python.org*. Python Software Foundation. . Retrieved 2009-06-27.
- [6] "The Making of Python" (<http://www.artima.com/intv/pythonP.html>). Artima Developer. . Retrieved 2007-03-22.
- [7] "A Brief Timeline of Python" (<http://python-history.blogspot.com/2009/01/brief-timeline-of-python.html>). Guido van Rossum. . Retrieved 2009-01-20.
- [8] van Rossum, Guido. "[Python-Dev] SETL (was: Lukewarm about range literals)" (<http://mail.python.org/pipermail/python-dev/2000-August/008881.html>). . Retrieved 2009-06-27.
- [9] "Why was Python created in the first place?" (<http://www.python.org/doc/faq/general/#why-was-python-created-in-the-first-place>). Python FAQ. . Retrieved 2007-03-22.
- [10] A.M. Kuchling and Moshe Zadka. "What's New in Python 2.0" (<http://www.amk.ca/python/2.0/>). . Retrieved 2007-03-22.
- [11] "Python 3.0 Release" (<http://python.org/download/releases/3.0/>). Python Software Foundation. . Retrieved 2009-07-08.
- [12] van Rossum, Guido (5 April 2006). "PEP 3000 -- Python 3000" (<http://www.python.org/dev/peps/pep-3000/>). Python Software Foundation. . Retrieved 2009-06-27.
- [13] The Cain Gang Ltd.. "Python Metaclasses: Who? Why? When?" (<http://www.python.org/community/pycon/dc2004/papers/24/metaclasses-pycon.pdf>) (PDF). . Retrieved 2009-06-27.

- [14] "3.3. Special method names" (<http://docs.python.org/3.0/reference/datamodel.html#special-method-names>). *The Python Language Reference*. Python Software Foundation. . Retrieved 2009-06-27.
- [15] Contracts for Python (<http://www.nongnu.org/pydbc/>), PyDBC
- [16] Contracts for Python (<http://www.wayforward.net/pycontract/>), pycontract
- [17] "6.5 itertools - Functions creating iterators for efficient looping" (<http://docs.python.org/lib/module-itertools.html>). Docs.python.org. . Retrieved 2008-11-24.
- [18] "PEP 20 - The Zen of Python" (<http://www.python.org/dev/peps/pep-0020/>). Python Software Foundation. 2004-08-23. . Retrieved 2008-11-24.
- [19] Python Culture (<http://www.python.org/dev/culture/>)
- [20] Python is... slow? (<http://peter.mapledesign.co.uk/weblog/archives/python-is-slow>) December 21st, 2004 — Peter Bowyer's weblog]
- [21] Python Patterns - An Optimization Anecdote (<http://www.python.org/doc/essays/list2str.html>)
- [22] Python Tutorial (<http://docs.python.org/tut/node3.html>)
- [23] Python Challenge tutorial (<http://www.pythonchallenge.com/>)
- [24] David Goodger. "Code Like a Pythonista: Idiomatic Python" (<http://python.net/~goodger/projects/pycon/2007/idiomatic/handout.html>). .; "How to think like a Pythonista" (<http://python.net/crew/mwh/hacks/objectthink.html>). .
- [25] Documentation of the PSP Scripting API can be found at *JASC Paint Shop Pro 9: Additional Download Resources* (<http://www.jasc.com/support/customer/articles/psp9components.asp>)
- [26] "About getting started with writing geoprocessing scripts" (http://webhelp.esri.com/arcgisdesktop/9.2/index.cfm?TopicName=About_getting_started_with_writing_geoprocessing_scripts). November 2006. . Retrieved April 2007.
- [27] Two video games using Python are "Civilization IV" (http://www.2kgames.com/civ4/blog_03.htm). 2kgames.com. . Retrieved 2008-11-24. and CCP. "EVE Online | EVE Insider | Dev Blog" (<http://myeve.eve-online.com/devblog.asp?a=blog&bid=488>). Myeve.eve-online.com. . Retrieved 2008-11-24.
- [28] ":: Pardus :: TÜBİTAK/UEKAE ::" (<http://www.pardus.org.tr/eng/projects/comar/PythonInPardus.html>). Pardus.org.tr. . Retrieved 2008-11-24.
- [29] Products and discussion of this use of Python include "IMMUNITY : Knowing You're Secure" (<http://www.immunitysec.com/products-immdbg.shtml>). Immunitysec.com. . Retrieved 2008-11-24.; CORE Security Technologies' open source software repository (<http://oss.coresecurity.com/>); "Wapiti - Web application security auditor" (<http://wapiti.sourceforge.net/>). Wapiti.sourceforge.net. . Retrieved 2008-11-24.; "TAOF - theartoffuzzing.com - Home" (<http://www.theartoffuzzing.com/joomla/>). Theartoffuzzing.com. . Retrieved 2008-11-24.; "[Dailydave] RE: Network Exploitation Tools aka Exploitation Engines" (<http://fist.immunitysec.com/pipermail/dailydave/2004-September/000851.html>). Fist.immunitysec.com. . Retrieved 2008-11-24.
- [30] "Coder Who Says Py: YouTube runs on Python!" (<http://sayspy.blogspot.com/2006/12/youtube-runs-on-python.html>). Sayspy.blogspot.com. December 12, 2006. . Retrieved 2008-11-24.
- [31] Review of original BitTorrent software (<http://www.onlamp.com/pub/a/python/2003/7/17/pythonnews.html>) at O'Reilly Python Dev Center
- [32] "Quotes about Python" (<http://python.org/about/quotes/>). Python.org. . Retrieved 2008-11-24.
- [33] "Organizations Using Python" (<http://wiki.python.org/moin/OrganizationsUsingPython>). Python.org. . Retrieved 2009-01-15.
- [34] CERN Document Server: Record#974627: Python : the holy grail of programming (<http://cdsweb.cern.ch/record/974627?ln=no>)
- [35] "Python Success Stories" (<http://www.python.org/about/success/usa/>). Python.org. . Retrieved 2008-11-24.
- [36] Python Slithers into Systems by Darryl K. Taft (<http://www.eweek.com/c/a/Application-Development/Python-Slithers-into-Systems/>)
- [37] "What is Sugar? - Sugar Labs" (<http://sugarlabs.org/go/Sugar>). sugarlabs.org. 2008-05-10. . Retrieved 0r-2-11.
- [38] "Is Python a good language for beginning programmers?" (<http://www.python.org/doc/faq/general/#is-python-a-good-language-for-beginning-programmers>). *General Python FAQ*. Python Foundation. March 7, 2005. . Retrieved 2007-03-21.
- [39] Myths about indentation in Python (http://www.secnetix.de/~olli/Python/block_indentation.hawk)
- [40] van Rossum, Guido (February 9, 2006). "Language Design Is Not Just Solving Puzzles" (<http://www.artima.com/weblogs/viewpost.jsp?thread=147358>). *Artima forums*. Artima. . Retrieved 2007-03-21.
- [41] van Rossum, Guido; Phillip J. Eby (April 21, 2006). "Coroutines via Enhanced Generators" (<http://www.python.org/peps/pep-0342.html>). *Python Enhancement Proposals*. Python Foundation. . Retrieved 2007-03-21.
- [42] "Why must 'self' be used explicitly in method definitions and calls?" (<http://www.python.org/doc/faq/general/#why-must-self-be-used-explicitly-in-method-definitions-and-calls>). *Python FAQ*. Python Foundation. .
- [43] "PEP 7 - Style Guide for C Code" (<http://www.python.org/dev/peps/pep-0007/>). Python.org. . Retrieved 2008-11-24.
- [44] CPython byte code (<http://docs.python.org/lib/bytencodes.html>)
- [45] Python 2.5 internals (<http://www.troegeer.eu/teaching/pythonvm08.pdf>)
- [46] "O'Reilly - An Interview with Guido van Rossum" (http://www.oreilly.com/pub/a/oreilly/frank/rossum_1099.html). Oreilly.com. . Retrieved 2008-11-24.
- [47] ProjectPlan (<http://code.google.com/p/unladen-swallow/wiki/ProjectPlan>), Plans for optimizing Python - unladen-swallow
- [48] PyMite: Python-on-a-chip (<http://wiki.python.org/moin/PyMite>)
- [49] Introduction to Psyco (<http://psyco.sourceforge.net/introduction.html>)
- [50] PEP 1 -- PEP Purpose and Guidelines (<http://www.python.org/dev/peps/pep-0001/>)
- [51] "Parade of the PEPs" (<http://www.python.org/doc/essays/pepparade.html>). Python.org. . Retrieved 2008-11-24.

- [52] Cannon, Brett. "Guido, Some Guys, and a Mailing List: How Python is Developed" (<http://python.org/dev/intro/>). *python.org*. Python Software Foundation. . Retrieved 2009-06-27.
 - [53] Norwitz, Neal (8 April 2002]]. "[Python-Dev] Release Schedules (was Stability & change)" (<http://mail.python.org/pipermail/python-dev/2002-April/022739.html>). . Retrieved 2009-06-27.
 - [54] Baxter, Anthony; Aahz (2001-03-15). "PEP 6 -- Bug Fix Releases" (<http://python.org/dev/peps/pep-0006/>). Python Software Foundation. . Retrieved 2009-06-27.
 - [55] Python Buildbot (<http://python.org/dev/buildbot/>), Python
 - [56] Przemyslaw Piotrowski, Build a Rapid Web Development Environment for Python Server Pages and Oracle (<http://www.oracle.com/technology/pub/articles/piotrowski-pythoncore.html>), Oracle Technology Network, July 2006. Retrieved October 21, 2008.
 - [57] "About Python" (<http://www.python.org/about/>). *python.org*. Python Software Foundation. . Retrieved 2009-06-27.
 - [58] "PEP 327 - Decimal Data Type" (<http://www.python.org/peps/pep-0327.html>). Python.org. . Retrieved 2008-11-24.
 - [59] <http://www.python.org/dev/peps/pep-0333/>
 - [60] "BOO - Gotchas for Python Users" (<http://boo.codehaus.org/Gotchas+for+Python+Users>). Boo.codehaus.org. . Retrieved 2008-11-24.
 - [61] "Cobra - Acknowledgements" (<http://cobra-language.com/docs/acknowledgements/>). cobra-language.com. . Retrieved 2010-04-07.
 - [62] "Cobra - Comparison to Python" (<http://cobra-language.com/docs/python/>). cobra-language.com. . Retrieved 2010-04-07.
 - [63] "proposals:iterators_and_generators [ES4 Wiki]" (http://wiki.ecmascript.org/doku.php?id=proposals:iterators_and_generators). Wiki.ecmascript.org. . Retrieved 2008-11-24.
 - [64] Kincaid, Jason (2009-11-10). "Google's Go: A New Programming Language That's Python Meets C++" (<http://www.techcrunch.com/2009/11/10/google-go-language/>). TechCrunch. . Retrieved 2010-01-29.
 - [65] James Strachan (2003-08-29). "Groovy - the birth of a new dynamic language for the Java platform" (<http://radio.weblogs.com/0112098/2003/08/29.html>). .
 - [66] Mike Lin. ""The Whitespace Thing" for OCaml" (<http://people.csail.mit.edu/mikelin/ocaml+twit/>). . Retrieved 2009-04-12.
 - [67] "TIP #3: TIP Format" (<http://www.tcl.tk/cgi-bin/tct/tip/3.html>). Tcl.tk. . Retrieved 2008-11-24.
 - [68] EEP - Erlang Enhancement Proposal (<http://www.erlang.org/eeps/eep-0001.html>)
 - [69] <http://www.qtrac.eu/py3book.html>
 - [70] <http://www.computerworld.com.au/index.php/id:66665771>
 - [71] <http://diveintopython.org/toc/index.html>
 - [72] <http://www.greenteapress.com/thinkpython/html/index.html>
 - [73] <http://www.dmoz.org/Computers/Programming/Languages/Python/>
-

Sage (mathematics software)

| | |
|---|---|
|  | |
| Initial release | 24 February 2005 |
| Stable release | 4.4.4 / June 23, 2010 |
| Written in | Python, Cython |
| Operating system | Cross-platform |
| Platform | Python |
| Type | Computer algebra system |
| License | GNU General Public License |
| Website | www.sagemath.org ^[1] |

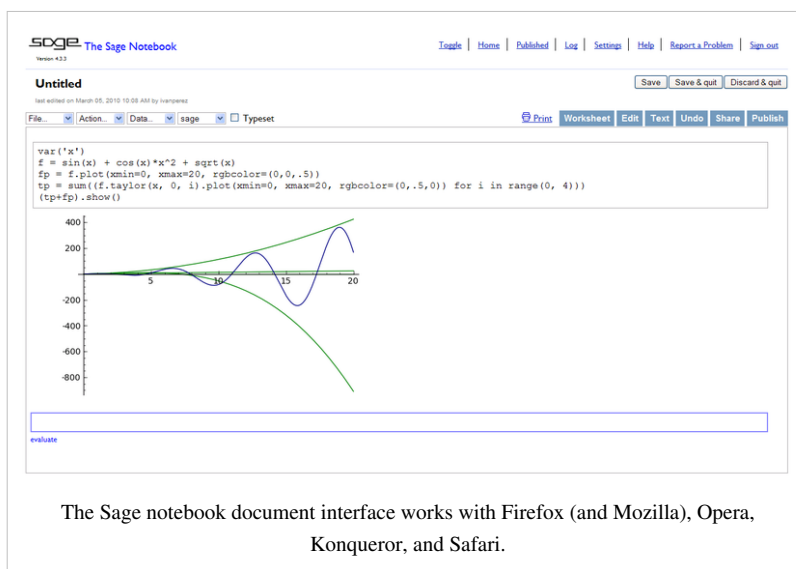
Sage is a software application which covers many aspects of mathematics, including algebra, combinatorics, numerical mathematics, and calculus.

The first version of Sage was released on 24 February 2005 as free and open source software under the terms of the GNU General Public License, with the initial goals of creating an "open source alternative to Magma, Maple, Mathematica, and MATLAB."^[2] The lead developer of Sage, William Stein, is a mathematician at the University of Washington.

Features

Some of the many features of Sage include^[3].

- A notebook document interface, for review and re-use of previous inputs and outputs, including graphics and text annotations usable from most web browsers including Firefox, Opera, Konqueror, and Safari. A secure connection via HTTPS to the notebook is supported when security or confidentiality are important, and allows Sage to be used both locally and remotely.
- A text-based command line interface using IPython
- The Python programming language supporting procedural, functional and object oriented constructs.



- Support for parallel processing using both multi-core processors found in many modern computers, multiple processors, in addition to distributed computing.
- Calculus using Maxima and SymPy
- Numerical Linear Algebra using the GSL, SciPy and NumPy.
- Libraries of elementary and special mathematical functions
- 2D and 3D graphs of both functions and data.
- Matrix and data manipulation tools including support for sparse arrays
- Multivariate statistics libraries, using the functionality of R and SciPy.
- A toolkit for adding user interfaces to calculations and applications.^[4]
- Tools for image processing using Pylab as well as the Python programming language.
- Graph theory visualization and analysis tools
- Libraries of number theory functions
- Import and export filters for data, images, video, sound, CAD, GIS, document and biomedical formats
- Support for complex number, arbitrary precision and symbolic computation for functions where this is appropriate.
- Technical word processing including formula editing and the ability to embed Sage inside LaTeX documents^[5]
- Network tools for connecting to SQL, Java, .NET, C++, FORTRAN provide by Twisted, This supports a large number of protocols including HTTP, NNTP, IMAP, SSH, IRC, FTP and others.
- Interfaces to some third-party software like Mathematica, Magma, and Maple, which allows users to combine software and compare output and performance. It is thus also a "front-end" to other mathematical tools similar to GNU TeXmacs.
- MoinMoin as a Wiki system for knowledge management.
- Documentation using Sphinx.
- An automated test-suite, which allows for testing on an end-user's computer.

Although not provided by Sage directly, Sage can be called from within Mathematica.^[6] A Mathematica notebook is available for this.^[7]

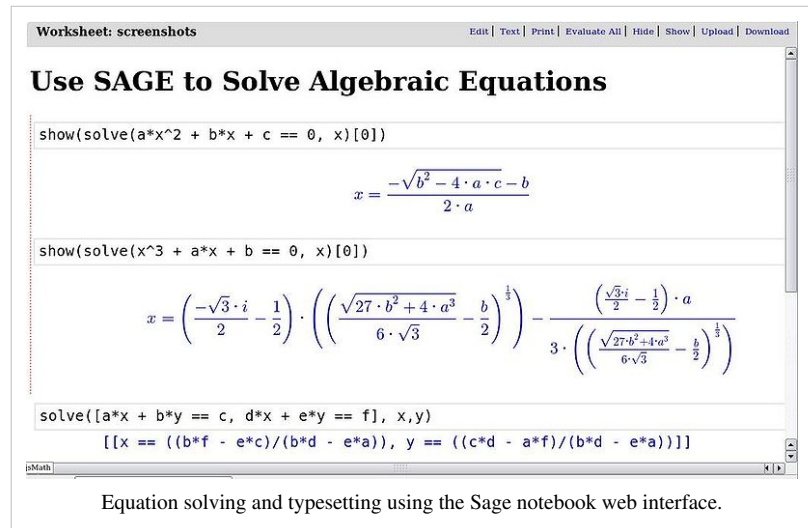
Design Philosophy of Sage

William Stein realized several important facts when designing Sage.

- To create viable alternatives to Magma, Maple, Mathematica, and MATLAB, would take hundreds, or thousands of man-years if one started from the beginning.
- There was a large collection of open-source mathematics software already written, but which was written in different languages (C, C++, Fortran and Python being the most common).

So rather than start from the beginning, Sage (which is written in Python and Cython) integrates all their specialized mathematics software into a common interface. A user needs to know only Python.

Where no open-source option exists for a particular class of problem, then this would be written in Sage. But Sage does not reinvent the wheel. The same design philosophy is used in commercial mathematics program such as Mathematica, but Sage can use a much wider range of open source software solutions than nonfree software, since some open source licensing imposes restrictions on commercial use of code.



Sage development uses both students and professionals for development. The development of Sage is supported by both volunteer work and grants.^[8]

Performance

Both binaries and source code are available for sage from the download page. If Sage is built from source code, many of the included libraries such as ATLAS, FLINT, and NTL will be tuned and optimized for that computer, taking into account the number of processors, the size of their caches, whether there is hardware support for SSE instructions etc.

Licensing and availability

Sage is free software, distributed under the terms of the GNU General Public License version 2+. Sage is available in many ways:

- The source code can be downloaded from the downloads page^[9]. Although not recommended for end users, development releases of Sage are also available.
- Binaries can be downloaded for Linux, OS X and Solaris (both x86 and SPARC), though the latest version of Sage does not build on Solaris x86, although it does on SPARC.
- A live CD containing a bootable Linux operating system is also available. This allows Sage to be tried, without being installed.
- Users can use an online version of Sage at sagenb.org^[10] or <http://t2nb.math.washington.edu:8000/>^[11], but with a limit to the amount of memory a user can use, as well as security restrictions.

Although Microsoft was sponsoring a native version of Sage for the Windows operating system^[12], users of Windows will currently have to use virtualization technology and run Sage under one of the aforementioned operating systems. The preferred system is VirtualBox.

Software packages contained in Sage

As stated above, the philosophy of Sage is to use existing open source libraries wherever they exist. Therefore it borrows from many projects in producing the final product.

Mathematics packages contained in Sage

| | |
|--------------------------------|--|
| Algebra | GAP, Maxima, Singular |
| Algebraic Geometry | Singular |
| Arbitrary Precision Arithmetic | GMP, MPFR, MPFI, NTL |
| Arithmetic Geometry | PARI/GP, NTL, mwrank, ecm |
| Calculus | Maxima, SymPy, GiNaC |
| Combinatorics | Symmetriza, Sage-Combinat |
| Linear Algebra | ATLAS, BLAS, LAPACK, NumPy, LinBox, IML, GSL |
| Graph Theory | NetworkX |
| Group Theory | GAP |
| Numerical computation | GSL, SciPy, NumPy, ATLAS |
| Number Theory | PARI/GP, FLINT, NTL |
| Statistical computing | R, SciPy |

Other packages contained in Sage

| | |
|----------------------------------|---------------------------------|
| Command line | IPython |
| Database | ZODB, Python Pickles, SQLite |
| Graphical Interface | Sage Notebook, jsmath |
| Graphics | Matplotlib, Tachyon3d, GD, Jmol |
| Interactive programming language | Python |
| Networking | Twisted |

Command interface examples

Calculus

```
x,a,b,c = var('x,a,b,c')

log(sqrt(a)).simplify_log() # returns log(a)/2
log(a/b).simplify_log() # returns log(a) - log(b)
sin(a+b).simplify_trig() # returns cos(a)*sin(b) + sin(a)*cos(b)
cos(a+b).simplify_trig() # returns cos(a)*cos(b) - sin(a)*sin(b)
(a+b)^5 # returns (b + a)^5
expand((a+b)^5) # returns b^5 + 5*a*b^4 + 10*a^2*b^3 +
# 10*a^3*b^2 + 5*a^4*b + a^5

limit((x^2+1)/(2+x+3*x^2), x=infinity) # returns 1/3
limit(sin(x)/x, x=0) # returns 1

diff(acos(x),x) # returns -1/sqrt(1 - x^2)
f = exp(x)*log(x)
f.diff(x,3) # returns e^x*log(x) + 3*e^x/x - 3*e^x/x^2 + 2*e^x/x^3

solve(a*x^2 + b*x + c, x) # returns [x == (-sqrt(b^2 - 4*a*c) -
b)/(2*a),
# x == (sqrt(b^2 - 4*a*c) - b)/(2*a)]

f = x^2 + 432/x
solve(f.diff(x)==0,x) # returns [x == 3*sqrt(3)*I - 3,
# x == -3*sqrt(3)*I - 3, x == 6]
```

Differential equations

```
t = var('t') # define a variable t
x = function('x',t) # define x to be a function of that variable
DE = lambda y: diff(y,t) + y - 1
desolve(DE(x(t)), [x,t]) # returns '%e^-t*(%e^t+%c)'
```

Linear algebra

```
A = Matrix([[1,2,3],[3,2,1],[1,1,1]])
y = vector([0,-4,-1])
A.solve_right(y) # returns (-2, 1, 0)
A.eigenvalues() # returns [5, 0, -1]

B = Matrix([[1,2,3],[3,2,1],[1,2,1]])
B.inverse() # returns [ 0 1/2 -1/2]
# [-1/4 -1/4 1]
# [ 1/2 0 -1/2]

# Call numpy for the Moore-Penrose pseudo-inverse,
# since Sage does not support that yet.

import numpy
C = Matrix([[1 , 1], [2 , 2]])
matrix(numpy.linalg.pinv(C.numpy())) # returns [0.1 0.2]
# [0.1 0.2]
```

Number Theory

```
prime_pi(1000000) # returns 78498, the number of primes less than one
million

E = EllipticCurve('389a') # construct an elliptic curve from its
Cremona label
P, Q = E.gens()
7*P + Q # returns (2869/676 : -171989/17576 : 1)
```

History

Only the major releases are listed below. Sage practices the "release early, release often" concept, with releases every two to three weeks.

Sage versions

| Version | Release Date | Description |
|-----------|--------------------------|--|
| 0.1 | January, 2005 | Included Pari, but not GAP or Singular |
| 0.2 - 0.4 | March to July 2005 | Cremona's database, multivariate polynomials, large finite fields and more documentation |
| 0.5 - 0.7 | August to September 2005 | Vector spaces, rings, modular symbols, and windows usage |
| 0.8 | October 2005 | Full distribution of GAP, Singular |
| 0.9 | November, 2005 | Maxima and clisp added |
| 1.0 | February, 2006 | |
| 2.0 | January, 2007 | |
| 3.0 | April, 2008 | |
| 4.0 | May, 2009 | |
| 5.0 | future | 5.0 Milestone ^[13] |

In 2007, Sage won first prize in the scientific software division of Les Trophées du Libre, an international competition for free software. ^[14]

See also

- Comparison of computer algebra systems
- Comparison of statistical packages
- Comparison of numerical analysis software

External links

- Project home page ^[1]
- Official Sage Documentation Manual, Reference, Tutorials, ... ^[15]
- Sage introduction videos ^[16]
- Use Sage online in your web-browser ^[10]
- Free software brings affordability, transparency to mathematics ^[17]
- AMS Notices Opinion - Open Source Mathematical Software ^[18]
- W. Stein's blog post on history of Sage ^[19]

References

- [1] <http://www.sagemath.org/>
- [2] Stein, William (2007-06-12). "SAGE Days 4" (<http://www.sagemath.org/why/stein-sd4.pdf>). . Retrieved 2007-08-02.
- [3] Sage documentation (<http://www.sagemath.org/help.html>)
- [4] "Sage Interact functionality" (<http://wiki.sagemath.org/interact>). . Retrieved 2008-04-11.
- [5] The TeX Catalogue OnLine, Entry for sagemath, Ctan Edition (<http://www.ctan.org/tex-archive/help/Catalogue/entries/sagemath.html>)
- [6] <http://facstaff.unca.edu/mcmclur/Mathematica/Sage/CallingSagefromMathematica>
- [7] <http://facstaff.unca.edu/mcmclur/Mathematica/Sage/UsingSage.nb> A Mathematica notebook to call Sage from Mathematica.
- [8] "Explicit Approaches to Modular Forms and Modular Abelian Varieties" (<http://www.nsf.gov/awardsearch/showAward.do?AwardNumber=0555776>). National Science Foundation. 2006-04-14. . Retrieved 2007-07-24.
- [9] <http://www.sagemath.org/download.html>
- [10] <http://www.sagenb.org/>
- [11] <http://t2nb.math.washington.edu:8000/>
- [12] Sage - Acknowledgment (<http://www.sagemath.org/development-ack.html>)
- [13] http://trac.sagemath.org/sage_trac/milestone/sage-5.0

-
- [14] "Free Software Brings Affordability, Transparency To Mathematics" (<http://www.sciencedaily.com/releases/2007/12/071206145213.htm>). Science Daily. December 7, 2007. . Retrieved 2008-07-20.
 - [15] <http://www.sagemath.org/doc/index.html>
 - [16] <http://www.sagemath.org/help-video.html>
 - [17] <http://www.physorg.com/news116173009.html>
 - [18] <http://www.ams.org/notices/200710/tx071001279p.pdf>
 - [19] <http://sagemath.blogspot.com/2009/12/mathematical-software-and-me-very.html>
-

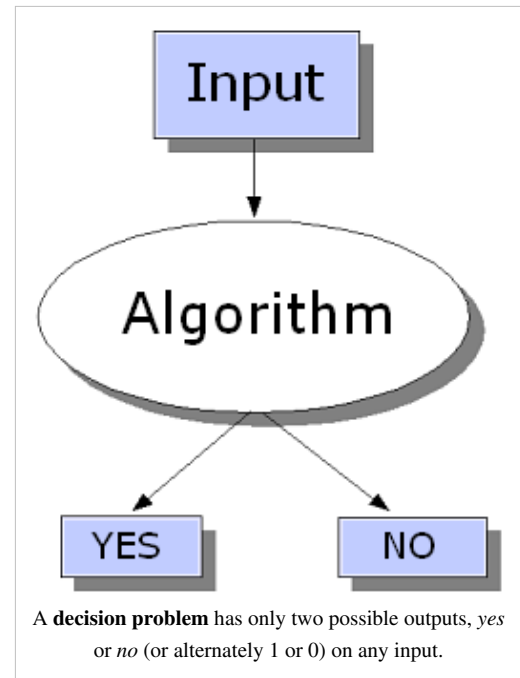
Decision procedures and implementations

Decision problem

In computability theory and computational complexity theory, a **decision problem** is a question in some formal system with a yes-or-no answer, depending on the values of some input parameters. For example, the problem "given two numbers x and y , does x evenly divide y ?" is a decision problem. The answer can be either 'yes' or 'no', and depends upon the values of x and y .

Decision problems are closely related to function problems, which can have answers that are more complex than a simple 'yes' or 'no'. A corresponding function problem is "given two numbers x and y , what is x divided by y ?". They are also related to optimization problems, which are concerned with finding the *best* answer to a particular problem.

A method for solving a decision problem given in the form of an algorithm is called a **decision procedure** for that problem. A decision procedure for the decision problem "given two numbers x and y , does x evenly divide y ?" would give the steps for determining whether x evenly divides y , given x and y . One such algorithm is long division, taught to many school children. If the remainder is zero the answer produced is 'yes', otherwise it is 'no'. A decision problem which can be solved by an algorithm, such as this example, is called **decidable**.



The field of computational complexity categorizes *decidable* decision problems by how difficult they are to solve. "Difficult", in this sense, is described in terms of the computational resources needed by the most efficient algorithm for a certain problem. The field of recursion theory, meanwhile, categorizes *undecidable* decision problems by Turing degree, which is a measure of the noncomputability inherent in any solution.

Research in computability theory has typically focused on decision problems. As explained in the section Equivalence with function problems below, there is no loss of generality.

Definition

A *decision problem* is any arbitrary yes-or-no question on an infinite set of inputs. Because of this, it is traditional to define the decision problem equivalently as: the set of inputs for which the problem returns yes.

These inputs can be natural numbers, but also other values of some other kind, such as strings of a formal language. Using some encoding, such as Gödel numberings, the strings can be encoded as natural numbers. Thus, a decision problem informally phrased in terms of a formal language is also equivalent to a set of natural numbers. To keep the formal definition simple, it is phrased in terms of subsets of the natural numbers.

Formally, a **decision problem** is a subset of the natural numbers. The corresponding informal problem is that of deciding whether a given number is in the set.

Examples

A classic example of a decidable decision problem is the set of prime numbers. It is possible to effectively decide whether a given natural number is prime by testing every possible nontrivial factor. Although much more efficient methods of primality testing are known, the existence of any effective method is enough to establish decidability.

Decidability

A decision problem A is called **decidable** or **effectively solvable** if A is a recursive set. A problem is called **partially decidable**, **semidecidable**, **solvable**, or **provable** if A is a recursively enumerable set. Partially decidable problems and any other problems that are not decidable are called **undecidable**.

The halting problem is an important undecidable decision problem; for more examples, see list of undecidable problems.

Complete problems

Decision problems can be ordered according to many-one reducibility and related feasible reductions such as Polynomial-time reductions. A decision problem P is said to be **complete** for a set of decision problems S if P is a member of S and every problem in S can be reduced to P . Complete decision problems are used in computational complexity to characterize complexity classes of decision problems. For example, the Boolean satisfiability problem is complete for the class NP of decision problems under polynomial-time reducibility.

History

The *Entscheidungsproblem*, German for "Decision-problem", is attributed to David Hilbert: "At [the] 1928 conference Hilbert made his questions quite precise. First, was mathematics *complete*... Second, was mathematics *consistent*... And thirdly, was mathematics *decidable*? By this he meant, did there exist a definite method which could, in principle be applied to any assertion, and which was guaranteed to produce a correct decision on whether that assertion was true" (Hodges, p. 91). Hilbert believed that "in mathematics there is no ignorabimus" (Hodges, p. 91ff) meaning 'we will not know'. See David Hilbert and Halting Problem for more.

Equivalence with function problems

A function problem consists of a partial function f ; the informal "problem" is to compute the values of f on the inputs for which it is defined.

Every function problem can be turned into a decision problem; the decision problem is just the graph of the associated function. (The graph of a function f is the set of pairs (x,y) such that $f(x) = y$.) If this decision problem were effectively solvable then the function problem would be as well. This reduction does not respect computational complexity, however. For example, it is possible for the graph of a function to be decidable in polynomial time (in which case running time is computed as a function of the pair (x,y)) when the function is not computable in polynomial time (in which case running time is computed as a function of x alone). The function $f(x) = 2^x$ has this property.

Every decision problem can be converted into the function problem of computing the characteristic function of the set associated to the decision problem. If this function is computable then the associated decision problem is decidable. However, this reduction is more liberal than the standard reduction used in computational complexity (sometimes called polynomial-time many-one reduction); for example, the complexity of the characteristic functions of an NP-complete problem and its co-NP-complete complement is exactly the same even though the underlying decision problems may not be considered equivalent in some typical models of computation.

Practical decision

Having practical decision procedures for classes of logical formulas is of considerable interest for program verification and circuit verification. Pure Boolean logical formulas are usually decided using SAT-solving techniques based on the DPLL algorithm. Conjunctive formulas over linear real or rational arithmetic can be decided using the Simplex algorithm, formulas in linear integer arithmetic (Presburger arithmetic) can be decided using Cooper's algorithm or William Pugh's Omega test. Formulas with negations, conjunctions and disjunctions combine the difficulties of satisfiability testing with that of decision of conjunctions; they are generally decided nowadays using SMT-solving technique, which combine SAT-solving with decision procedures for conjunctions and propagation techniques. Real polynomial arithmetic, also known as the theory of real closed fields, is decidable, for instance using the Cylindrical algebraic decomposition; unfortunately the complexity of that algorithm is excessive for most practical uses.

A leading scientific conference in this field is CAV.

See also

- ALL (complexity)
- Decidability (logic) – for the problem of deciding whether a formula is a consequence of a logical theory.
- yes-no question
- Optimization problem
- Search problem
- Counting problem (complexity)
- Function problem

References

- Hanika, Jiri. *Search Problems and Bounded Arithmetic*. PhD Thesis, Charles University, Prague. http://www.eccc.uni-trier.de/static/books/Search_Problems_and_Bounded_Arithmetic/
- Hodges, A., *Alan Turing: The Enigma*, Simon and Schuster, New York. Cf Chapter "The Spirit of Truth" for some more history that led to Turing's work.

Hodges references a biography of David Hilbert: Constance Reid, *Hilbert* (George Allen & Unwin; Springer-Verlag, 1970). There are apparently more recent editions.

- Kozen, D.C. (1997), *Automata and Computability*, Springer.
- Hartley Rogers, Jr., *The Theory of Recursive Functions and Effective Computability*, MIT Press, ISBN 0-262-68052-1 (paperback), ISBN 0-07-053522-1
- Sipser, M. (1996), *Introduction to the Theory of Computation*, PWS Publishing Co.
- Robert I. Soare (1987), *Recursively Enumerable Sets and Degrees*, Springer-Verlag, ISBN 0-387-15299-7

Effective decision

- Daniel Kroening & Ofer Strichman, *Decision procedures*, Springer, ISBN 978-3-540-74104-6
- Aaron Bradley & Zohar Manna, *The calculus of computation*, Springer, ISBN 978-3-540-74112-1

Boolean satisfiability problem

Satisfiability is the problem of determining if the variables of a given Boolean formula can be assigned in such a way as to make the formula evaluate to TRUE. Equally important is to determine whether no such assignments exist, which would imply that the function expressed by the formula is identically FALSE for all possible variable assignments. In this latter case, we would say that the function is unsatisfiable; otherwise it is satisfiable. To emphasize the binary nature of this problem, it is frequently referred to as *Boolean* or *propositional satisfiability*. The shorthand "**SAT**" is also commonly used to denote it, with the implicit understanding that the function and its variables are all binary-valued.

Basic definitions, terminology and applications

In complexity theory, the **satisfiability problem** (SAT) is a decision problem, whose instance is a Boolean expression written using only AND, OR, NOT, variables, and parentheses. The question is: given the expression, is there some assignment of *TRUE* and *FALSE* values to the variables that will make the entire expression true? A formula of propositional logic is said to be *satisfiable* if logical values can be assigned to its variables in a way that makes the formula true. The boolean satisfiability problem is NP-complete. The propositional satisfiability problem (PSAT), which decides whether a given propositional formula is satisfiable, is of central importance in various areas of computer science, including theoretical computer science, algorithmics, artificial intelligence, hardware design, electronic design automation, and verification.

A **literal** is either a variable or the negation of a variable (the negation of an expression can be reduced to negated variables by De Morgan's laws). For example, x_1 is a **positive literal** and $\text{not}(x_2)$ is a **negative literal**.

A **clause** is a disjunction of literals. For example, $x_1 \vee \text{not}(x_2)$ is a clause.

There are several special cases of the Boolean satisfiability problem in which the formulae are required to be conjunctions of clauses (i.e. formulae in conjunctive normal form). Determining the satisfiability of a formula in conjunctive normal form where each clause is limited to at most three literals is NP-complete; this problem is called "3SAT", "3CNFSAT", or "3-satisfiability". Determining the satisfiability of a formula in which each clause is limited to at most two literals is NL-complete; this problem is called "2SAT". Determining the satisfiability of a formula in which each clause is a Horn clause (i.e. it contains at most one positive literal) is P-complete; this problem is called Horn-satisfiability.

The Cook–Levin theorem proves that the Boolean satisfiability problem is NP-complete, and in fact, this was the first decision problem proved to be NP-complete. However, beyond this theoretical significance, efficient and scalable algorithms for SAT that were developed over the last decade have contributed to dramatic advances in our ability to automatically solve problem instances involving tens of thousands of variables and millions of constraints. Examples of such problems in electronic design automation (EDA) include formal equivalence checking, model checking, formal verification of pipelined microprocessors, automatic test pattern generation, routing of FPGAs, and so on. A SAT-solving engine is now considered to be an essential component in the EDA toolbox.

Complexity and restricted versions

NP-completeness

SAT was the first known NP-complete problem, as proved by Stephen Cook in 1971 (see Cook's theorem for the proof). Until that time, the concept of an NP-complete problem did not even exist. The problem remains NP-complete even if all expressions are written in *conjunctive normal form* with 3 variables per clause (3-CNF), yielding the **3SAT** problem. This means the expression has the form:

$$\begin{aligned} &(x_{11} \text{ OR } x_{12} \text{ OR } x_{13}) \text{ AND} \\ &(x_{21} \text{ OR } x_{22} \text{ OR } x_{23}) \text{ AND} \\ &(x_{31} \text{ OR } x_{32} \text{ OR } x_{33}) \text{ AND } \dots \end{aligned}$$

where each x is a variable or a negation of a variable, and each variable can appear multiple times in the expression.

A useful property of Cook's reduction is that it preserves the number of accepting answers. For example, if a graph has 17 valid 3-colorings, the SAT formula produced by the reduction will have 17 satisfying assignments.

NP-completeness only refers to the run-time of the worst case instances. Many of the instances that occur in practical applications can be solved much more quickly. See runtime behavior below.

SAT is easier if the formulas are restricted to those in disjunctive normal form, that is, they are disjunction (OR) of terms, where each term is a conjunction (AND) of literals (possibly negated variables). Such a formula is indeed satisfiable if and only if at least one of its terms is satisfiable, and a term is satisfiable if and only if it does not contain both x and NOT x for some variable x . This can be checked in polynomial time.

2-satisfiability

SAT is also easier if the number of literals in a clause is limited to 2, in which case the problem is called 2SAT. This problem can also be solved in polynomial time, and in fact is complete for the class NL. Similarly, if we limit the number of literals per clause to 2 and change the AND operations to XOR operations, the result is *exclusive-or 2-satisfiability*, a problem complete for SL = L.

One of the most important restrictions of SAT is HORNSAT, where the formula is a conjunction of Horn clauses. This problem is solved by the polynomial-time Horn-satisfiability algorithm, and is in fact P-complete. It can be seen as P's version of the Boolean satisfiability problem.

Provided that the complexity classes P and NP are not equal, none of these restrictions are NP-complete, unlike SAT. The assumption that P and NP are not equal is currently not proven.

3-satisfiability

3-satisfiability is a special case of k -satisfiability (k -SAT) or simply satisfiability (SAT), when each clause contains exactly $k = 3$ literals. It was one of Karp's 21 NP-complete problems.

Here is an example, where \neg indicates negation:

$$E = (x_1 \text{ OR } \neg x_2 \text{ OR } \neg x_3) \text{ AND } (x_1 \text{ OR } x_2 \text{ OR } x_4)$$

E has two clauses (denoted by parentheses), four variables (x_1, x_2, x_3, x_4), and $k=3$ (three literals per clause).

To solve this instance of the decision problem we must determine whether there is a truth value (TRUE or FALSE) we can assign to each of the literals (x_1 through x_4) such that the entire expression is TRUE. In this instance, there is such an assignment ($x_1 = \text{TRUE}, x_2 = \text{TRUE}, x_3 = \text{TRUE}, x_4 = \text{TRUE}$), so the answer to this instance is YES. This is one of many possible assignments, with for instance, any set of assignments including $x_1 = \text{TRUE}$ being sufficient. If there were no such assignment(s), the answer would be NO.

Since k -SAT (the general case) reduces to 3-SAT, and 3-SAT can be proven to be NP-complete, it can be used to prove that other problems are also NP-complete. This is done by showing how a solution to another problem could

be used to solve 3-SAT. An example of a problem where this method has been used is "Clique". It's often easier to use reductions from 3-SAT than SAT to problems that researchers are attempting to prove NP-complete.

3-SAT can be further restricted to One-in-three 3SAT, where we ask if *exactly* one of the variables in each clause is true, rather than *at least* one. One-in-three 3SAT remains NP-complete.

Horn-satisfiability

A clause is Horn if it contains at most one positive literal. Such clauses are of interest because they are able to express implication of one variable from a set of other variables. Indeed, one such clause $\neg x_1 \vee \dots \vee \neg x_n \vee y$ can be rewritten as $x_1 \wedge \dots \wedge x_n \rightarrow y$, that is, if x_1, \dots, x_n are all true, then y needs to be true as well.

The problem of deciding whether a set of Horn clauses is satisfiable is in P. This problem can indeed be solved by a single step of the Unit propagation, which produces the single minimal (w.r.t. the set of literal assigned to true) model of the set of Horn clauses.

A generalization of the class of Horn formulae is that of renamable-Horn formulae, which is the set of formulae that can be placed in Horn form by replacing some variables with their respective negation. Checking the existence of such a replacement can be done in linear time; therefore, the satisfiability of such formulae is in P as it can be solved by first performing this replacement and then checking the satisfiability of the resulting Horn formula.

Schaefer's dichotomy theorem

The restrictions above (CNF, 2CNF, 3CNF, Horn) bound the considered formulae to be conjunction of subformulae; each restriction states a specific form for all subformulae: for example, only binary clauses can be subformulae in 2CNF.

Schaefer's dichotomy theorem states that, for any restriction to Boolean operators that can be used to form these subformulae, the corresponding satisfiability problem is in P or NP-complete. The membership in P of the satisfiability of 2CNF and Horn formulae are special cases of this theorem.

Runtime behavior

As mentioned briefly above, though the problem is NP-complete, many practical instances can be solved much more quickly. Many practical problems are actually "easy", so the SAT solver can easily find a solution, or prove that none exists, relatively quickly, even though the instance has thousands of variables and tens of thousands of constraints. Other much smaller problems exhibit run-times that are exponential in the problem size, and rapidly become impractical. Unfortunately, there is no reliable way to tell the difficulty of the problem without trying it. Therefore, almost all SAT solvers include time-outs, so they will terminate even if they cannot find a solution. Finally, different SAT solvers will find different instances easy or hard, and some excel at proving unsatisfiability, and others at finding solutions. All of these behaviors can be seen in the SAT solving contests.^[1]

Extensions of SAT

An extension that has gained significant popularity since 2003 is Satisfiability modulo theories (SMT) that can enrich CNF formulas with linear constraints, arrays, all-different constraints, uninterpreted functions, etc. Such extensions typically remain NP-complete, but very efficient solvers are now available that can handle many such kinds of constraints.

The satisfiability problem becomes more difficult (PSPACE-complete) if we extend our logic to include second-order Booleans, allowing *quantifiers* such as "for all" and "there exists" that bind the Boolean variables. An example of such an expression would be:

$$\forall x (\exists y (\exists z ((x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z)))) .$$

SAT itself uses only \exists quantifiers. If we allow only \forall quantifiers, it becomes the Co-NP-complete tautology problem. If we allow both, the problem is called the quantified Boolean formula problem (QBF), which can be shown to be PSPACE-complete. It is widely believed that PSPACE-complete problems are strictly harder than any problem in NP, although this has not yet been proved.

A number of variants deal with the number of variable assignments making the formula true. Ordinary SAT asks if there is at least one such assignment. MAJSAT, which asks if the majority of all assignments make the formula true, is complete for PP, a probabilistic class. The problem of how many variable assignments satisfy a formula, not a decision problem, is in #P. UNIQUE-SAT or USAT is the problem of determining whether a formula known to have either zero or one satisfying assignments has zero or has one. Although this problem seems easier, it has been shown that if there is a practical (randomized polynomial-time) algorithm to solve this problem, then all problems in NP can be solved just as easily.

The maximum satisfiability problem, an FNP generalization of SAT, asks for the maximum number of clauses which can be satisfied by any assignment. It has efficient approximation algorithms, but is NP-hard to solve exactly. Worse still, it is APX-complete, meaning there is no polynomial-time approximation scheme (PTAS) for this problem unless $P=NP$.

NP-Complete Solutions in SAT

Theorem (Karp-Lipton-Sipser): If $NP \subseteq P/poly$ then $PH = \Sigma_2^P$.

Note if $NP \subseteq P/poly$ then SAT has a polynomial-size circuit family.

Namely, for some $c > 0$ there is a sequence of circuits W_1, W_2, W_3, \dots , where W_n is a circuit of size n^c deciding satisfiability of all Boolean formulas whose encoding size is $\leq n$.

Of course, $NP \subseteq P/poly$ merely implies the existence of such a family; there may be no easy way to construct the circuits.

Lemma (Self-reducibility of SAT)

There is a polynomial-time computable function h such that if $\{W_n\}_{n \geq 1}$ is a circuit family that solves SAT, then for all Boolean formula ϕ : $\phi \in SAT$ iff $h(\phi, W|_{\phi})$ is a satisfying assignment for ϕ . (1)

Proof: The main idea is to use the circuit to generate a satisfying assignment as follows.

Ask the circuit if ϕ is satisfiable.

If so, ask it if $\phi(x_1 = T)$ (i.e., ϕ with the first variable assigned True) is satisfiable.

The circuit's answer allows us to reduce the size of the formula.

If the circuit says no, we can conclude $\phi(x_1 = F)$ is true, and have thus reduced the number of variables by 1.

If the circuit says yes, we can substitute $x_1 = T$ and again reduced the number of variables by 1.

Continuing this way, we can generate a satisfying assignment. This proves the Lemma. (Aside: The formal name for the above property of SAT is downward self-reducibility. All NP-complete languages have this property.) QED

Now we prove the Theorem.

We show $NP \subseteq P/poly$ implies $\Pi_2^P \subseteq \Sigma_2^P$. Let $L \in \Pi_2^P$. Then there is a language $L_1 \in NP$ and $c > 0$ such that $L = \{x \in \{0, 1\}^* : \forall y, |y| \leq |x|^c, (x, y) \in L_1\}$. (2)

Since $L_1 \in NP$, there is a polynomial-time reduction, say g , from L_1 to SAT.

Thus $\forall z \in \{0, 1\}^* : z \in L_1$ iff $g(z) \in SAT$.

Suppose further that $d > 0$ is such that $|g(z)| \leq |z|^d$.

Now we can rewrite (2) as $L = \{x \in \{0, 1\}^* : \forall y, |y| \leq |x|^c, g(x, y) \in SAT\}$.

Note that $|g(x, y)| \leq (|x| + |y|)d$.

Let us simplify this as $|x|cd$.

Thus if $W_{n \geq 1}$ is a nk -sized circuit family for SAT, then by 1) we have $L = \{x : \forall y, |y| \leq |x|c1, h(g(x, y), W|x|cd) \text{ is a satisfying assignment for } g(x, y)\}$.

Now we are almost done.

Even though there may be no way for us to construct the circuit for SAT, we can just try to "guess" it.

Namely, an input x is in L iff $\exists W$, a circuit with $|x|cd$ inputs and size $|x|cdk$ such that $\forall y, |y| \leq |x|c1, h(g(x, y), W)$ is a satisfying assignment for $g(x, y)$.

Since h is computable in polynomial time, we have thus shown $L \in \Sigma_p^2$. QED1[2]

Algorithms for solving SAT

There are two classes of high-performance algorithms for solving instances of SAT in practice: modern variants of the DPLL algorithm, such as Chaff, GRASP or march^[3]; and stochastic local search algorithms, such as WalkSAT.

A DPLL SAT solver employs a systematic backtracking search procedure to explore the (exponentially-sized) space of variable assignments looking for satisfying assignments. The basic search procedure was proposed in two seminal papers in the early 60s (see references below) and is now commonly referred to as the Davis-Putnam-Logemann-Loveland algorithm ("DPLL" or "DLL"). Theoretically, exponential lower bounds have been proved for the DPLL family of algorithms.

Modern SAT solvers (developed in the last ten years) come in two flavors: "conflict-driven" and "look-ahead". Conflict-driven solvers augment the basic DPLL search algorithm with efficient conflict analysis, clause learning, non-chronological backtracking (aka backjumping), as well as "two-watched-literals" unit propagation, adaptive branching, and random restarts. These "extras" to the basic systematic search have been empirically shown to be essential for handling the large SAT instances that arise in Electronic Design Automation (EDA). Look-ahead solvers have especially strengthened reductions (going beyond unit-clause propagation) and the heuristics, and they are generally stronger than conflict-driven solvers on hard instances (while conflict-driven solvers can be much better on large instances which have inside actually an easy instance).

Modern SAT solvers are also having significant impact on the fields of software verification, constraint solving in artificial intelligence, and operations research, among others. Powerful solvers are readily available as free and open source software. In particular, the conflict-driven MiniSAT^[4], which was relatively successful at the 2005 SAT competition^[5], only has about 600 lines of code. An example for look-ahead solvers is march_dl^[3], which won a prize at the 2007 SAT competition^[5].

Genetic algorithms and other general-purpose stochastic local search methods are also being used to solve SAT problems, especially when there is no or limited knowledge of the specific structure of the problem instances to be solved. Certain types of large random satisfiable instances of SAT can be solved by survey propagation (SP). Particularly in hardware design and verification applications, satisfiability and other logical properties of a given propositional formula are sometimes decided based on a representation of the formula as a binary decision diagram (BDD).

Propositional satisfiability has various generalisations, including satisfiability for quantified Boolean formula problem, for first- and second-order logic, constraint satisfaction problems, 0-1 integer programming, and maximum satisfiability problem.

Many other decision problems, such as graph coloring problems, planning problems, and scheduling problems, can be easily encoded into SAT.

See also

- Unsatisfiable core
- Satisfiability Modulo Theories

References

References are listed in order by date of publishing:

- M. Davis and H. Putnam, A Computing Procedure for Quantification Theory (doi:10.1145/321033.321034), Journal of the Association for Computing Machinery, vol. 7, no. 3, pp. 201–215, 1960.
- M. Davis, G. Logemann, and D. Loveland, A Machine Program for Theorem-Proving (doi:10.1145/368273.368557), Communications of the ACM, vol. 5, no. 7, pp. 394–397, 1962.
- S. A. Cook, The Complexity of Theorem Proving Procedures (doi:10.1145/800157.805047), in Proc. 3rd Ann. ACM Symp. on Theory of Computing, pp. 151–158, Association for Computing Machinery, 1971.
- Michael R. Garey and David S. Johnson (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman. ISBN 0-7167-1045-5. A9.1: LO1 – LO7, pp. 259 – 260.
- J. P. Marques-Silva and K. A. Sakallah, GRASP: A Search Algorithm for Propositional Satisfiability (doi:10.1109/12.769433), IEEE Transactions on Computers, vol. 48, no. 5, pp. 506–521, 1999.
- J.-P. Marques-Silva and T. Glass, Combinational Equivalence Checking Using Satisfiability and Recursive Learning (doi:10.1109/DATE.1999.761110), in Proc. Design, Automation and Test in Europe Conference, pp. 145–149, 1999.
- R. E. Bryant, S. M. German, and M. N. Velez, Microprocessor Verification Using Efficient Decision Procedures for a Logic of Equality with Uninterpreted Functions ^[6], in Analytic Tableaux and Related Methods, pp. 1–13, 1999.
- M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, Chaff: engineering an efficient SAT solver (doi:10.1145/378239.379017), in Proc. 38th ACM/IEEE Design Automation Conference, pp. 530–535, Las Vegas, Nevada, 2001.
- E. Clarke ^[7], A. Biere ^[8], R. Raimi, and Y. Zhu, Bounded Model Checking Using Satisfiability Solving (doi:10.1023/A:1011276507260), Formal Methods in System Design, vol. 19, no. 1, 2001.
- M. Perkowski and A. Mishchenko, "Logic Synthesis for Regular Layout using Satisfiability," in Proc. Intl Workshop on Boolean Problems, 2002.
- G.-J. Nam, K. A. Sakallah, and R. Rutenbar, A New FPGA Detailed Routing Approach via Search-Based Boolean Satisfiability (doi:10.1109/TCAD.2002.1004311), IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 21, no. 6, pp. 674–684, 2002.
- N. Een and N. Sörensson, An Extensible SAT-solver (doi:10.1007/b95238), in Satisfiability Workshop, 2003.
- D. Babić, J. Bingham, and A. J. Hu, B-Cubing: New Possibilities for Efficient SAT-Solving (doi:10.1109/TC.2006.175), IEEE Transactions on Computers 55(11):1315–1324, 2006.
- C. Rodríguez, M. Villagra and B. Barán, Asynchronous team algorithms for Boolean Satisfiability (doi:10.1109/BIMNICS.2007.4610083), Bionetics2007, pp. 66–69, 2007.

[1] "The international SAT Competitions web page" (<http://www.satcompetition.org/>). . Retrieved 2007-11-15.

[2] http://groups.google.com/group/sci.math/browse_thread/thread/0e6b38c83b02e3c8#

[3] http://www.st.ewi.tudelft.nl/sat/march_dl.php

[4] <http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/>

[5] <http://www.satcompetition.org/>

[6] <http://portal.acm.org/citation.cfm?id=709275>

[7] <http://www.cs.cmu.edu/~emc/>

[8] <http://fmv.jku.at/biere/>

External links

More information on SAT:

- SAT and MAX-SAT for the Lay-researcher (<http://users.ecs.soton.ac.uk/mqq06r/sat/>)

SAT Applications:

- WinSAT v2.04 (<http://users.ecs.soton.ac.uk/mqq06r/winsat/>): A Windows-based SAT application made particularly for researchers.

SAT Solvers:

- Chaff (<http://www.princeton.edu/~chaff/>)
- HyperSAT (<http://www.domagoj-babic.com/index.php/ResearchProjects/HyperSAT>)
- Spear (<http://www.domagoj-babic.com/index.php/ResearchProjects/Spear>)
- The MiniSAT Solver (<http://minisat.se/>)
- UBCSAT (<http://www.satlib.org/ubcsat/>)
- Sat4j (<http://www.sat4j.org/>)
- RSat (<http://reasoning.cs.ucla.edu/rsat/home.html>)
- Fast SAT Solver (<http://dudka.cz/fss>) - simple but fast implementation of SAT solver based on genetic algorithms
- CVC3 (<http://www.cs.nyu.edu/acsys/cvc3/>)

Conferences/Publications:

- SAT 2009: Twelfth International Conference on Theory and Applications of Satisfiability Testing (<http://www.cs.swansea.ac.uk/~csoliver/SAT2009/>)
- SAT 2008: Eleventh International Conference on Theory and Applications of Satisfiability Testing (<http://wwwcs.uni-paderborn.de/cs/ag-klbue/en/workshops/sat-08/sat08-main.php>)
- SAT 2007: Tenth International Conference on Theory and Applications of Satisfiability Testing (<http://sat07.ecs.soton.ac.uk/>)
- Journal on Satisfiability, Boolean Modeling and Computation (<http://jsat.ewi.tudelft.nl>)
- Survey Propagation (<http://www.ictp.trieste.it/~zecchina/SP/>)

Benchmarks:

- Forced Satisfiable SAT Benchmarks (<http://www.nlsde.buaa.edu.cn/~kexu/benchmarks/benchmarks.htm>)
- IBM Formal Verification SAT Benchmarks (http://www.haifa.il.ibm.com/projects/verification/RB_Homepage/bmcbenchmarks.html)
- SATLIB (<http://www.satlib.org>)
- Software Verification Benchmarks (http://www.cs.ubc.ca/~babic/index_benchmarks.htm)
- Fadi Aloul SAT Benchmarks (<http://www.aloul.net/benchmarks.html>)

SAT solving in general:

- <http://www.satlive.org>
- <http://www.satisfiability.org>

Evaluation of SAT solvers:

- Yearly evaluation of SAT solvers (<http://www.maxsat.udl.cat/>)
 - SAT solvers evaluation results for 2008 (<http://www.maxsat.udl.cat/08/ms08.pdf>)
-

This article includes material from a column in the ACM SIGDA (<http://www.sigda.org>) e-newsletter (<http://www.sigda.org/newsletter/index.html>) by Prof. Karem Sakallah (<http://www.eecs.umich.edu/~karem>)
Original text is available here (http://www.sigda.org/newsletter/2006/eNews_061201.html)

Constraint satisfaction

In artificial intelligence and operations research, **constraint satisfaction** is the process of finding a solution to a set of constraints that impose conditions that the variables must satisfy. A solution is therefore a vector of variables that satisfies all constraints.

The techniques used in constraint satisfaction depend on the kind of constraints being considered. Often used are constraints on a finite domain, to the point that constraint satisfaction problems are typically identified with problems based on constraints on a finite domain. Such problems are usually solved via search, in particular a form of backtracking or local search. Constraint propagation are other methods used on such problems; most of them are incomplete in general, that is, they may solve the problem or prove it unsatisfiable, but not always. Constraint propagation methods are also used in conjunction with search to make a given problem simpler to solve. Other considered kinds of constraints are on real or rational numbers; solving problems on these constraints is done via variable elimination or the simplex algorithm.

Constraint satisfaction originated in the field of artificial intelligence in the 1970s (see for example (Laurière 1978)). During the 1980s and 1990s, embedding of constraints into a programming language were developed. Languages often used for constraint programming are Prolog and C++.

Constraint satisfaction problem

As originally defined in artificial intelligence, constraints enumerate the possible values a set of variables may take. Informally, a finite domain is a finite set of arbitrary elements. A constraint satisfaction problem on such domain contains a set of variables whose values can only be taken from the domain, and a set of constraints, each constraint specifying the allowed values for a group of variables. A solution to this problem is an evaluation of the variables that satisfies all constraints. In other words, a solution is a way for assigning a value to each variable in such a way that all constraints are satisfied by these values.

In practice, constraints are often expressed in compact form, rather than enumerating all values of the variables that would satisfy the constraint. One of the most used constraints is the one establishing that the values of the affected variables must be all different.

Problems that can be expressed as constraint satisfaction problems are the Eight queens puzzle, the Sudoku solving problem, the Boolean satisfiability problem, scheduling problems and various problems on graphs such as the graph coloring problem.

While usually not included in the above definition of a constraint satisfaction problem, arithmetic equations and inequalities bound the values of the variables they contain and can therefore be considered a form of constraints. Their domain is the set of numbers (either integer, rational, or real), which is infinite; therefore, the relations of these constraints may be infinite as well; for example, $X = Y + 1$ has an infinite number of pairs of satisfying values. Arithmetic equations and inequalities are often not considered within the definition of a "constraint satisfaction problem", which is limited to finite domains. They are however used often in constraint programming.

Solving

Constraint satisfaction problems on finite domains are typically solved using a form of search. The most used techniques are variants of backtracking, constraint propagation, and local search. These techniques are used on problems with nonlinear constraints.

Variable elimination and the simplex algorithm are used for solving linear and polynomial equations and inequalities, and problems containing variables with infinite domain. These are typically solved as optimization problems in which the optimized function is the number of violated constraints.

Complexity

Solving a constraint satisfaction problem on a finite domain is an NP complete problem. Research has shown a number of tractable subcases, some limiting the allowed constraint relations, some requiring the scopes of constraints to form a tree, possibly in a reformulated version of the problem. Research has also established relationship of the constraint satisfaction problem with problems in other areas such as finite model theory.

Constraint programming

Constraint programming is the use of constraints as a programming language to encode and solve problems. This is often done by embedding constraints into a programming language, which is called the host language. Constraint programming originated from a formalization of equalities of terms in Prolog II, leading to a general framework for embedding constraints into a logic programming language. The most common host languages are Prolog, C++, and Java, but other languages have been used as well.

Constraint logic programming

A constraint logic program is a logic program that contains constraints in the bodies of clauses. As an example, the clause $A(X):-X>0, B(X)$ is a clause containing the constraint $X>0$ in the body. Constraints can also be present in the goal. The constraints in the goal and in the clauses used to prove the goal are accumulated into a set called constraint store. This set contains the constraints the interpreter has assumed satisfiable in order to proceed in the evaluation. As a result, if this set is detected unsatisfiable, the interpreter backtracks. Equations of terms, as used in logic programming, are considered a particular form of constraints which can be simplified using unification. As a result, the constraint store can be considered an extension of the concept of substitution that is used in regular logic programming. The most common kinds of constraints used in constraint logic programming are constraints over integers/rational/real numbers and constraints over finite domains.

Concurrent constraint logic programming languages have also been developed. They significantly differ from non-concurrent constraint logic programming in that they are aimed at programming concurrent processes that may not terminate. Constraint handling rules can be seen as a form of concurrent constraint logic programming, but are also sometimes used within a non-concurrent constraint logic programming language. They allow for rewriting constraints or to infer new ones based on the truth of conditions.

Constraint satisfaction toolkits

Constraint satisfaction toolkits are software libraries for imperative programming languages that are used to encode and solve a constraint satisfaction problem.

- Cassowary constraint solver is an open source project for constraint satisfaction (accessible from C, Java, Python and other languages).
- Comet, a commercial programming language and toolkit
- Gecode, an open source portable toolkit written in C++ developed as a production-quality and highly efficient implementation of a complete theoretical background.
- JaCoP (solver) an open source Java constraint solver ^[1]
- Koalog ^[2] a commercial Java-based constraint solver.
- logilab-constraint ^[3] an open source constraint solver written in pure Python with constraint propagation algorithms.
- MINION ^[4] an open-source constraint solver written in C++, with a small language for the purpose of specifying models/problems.
- ZDC ^[5] is an open source program developed in the Computer-Aided Constraint Satisfaction Project ^[6] for modelling and solving constraint satisfaction problems.

Other constraint programming languages

Constraint toolkits are a way for embedding constraints into an imperative programming language. However, they are only used as external libraries for encoding and solving problems. An approach in which constraints are integrated into an imperative programming language is taken in the Kaleidoscope programming language.

Constraints have also been embedded into functional programming languages.

See also

- Constraint satisfaction problem
- Constraint (mathematics)
- Candidate solution
- Boolean satisfiability problem
- Decision theory

References

- Apt, Krzysztof (2003). *Principles of constraint programming*. Cambridge University Press. ISBN 0-521-82583-0.
- Dechter, Rina (2003). *Constraint processing* ^[7]. Morgan Kaufmann. ISBN 1-55860-890-7.
- Dincbas, M.; Simonis, H.; Van Hentenryck, P. (1990). "Solving Large Combinatorial Problems in Logic Programming". *Journal of logic programming* **8** (1-2): 75–93. doi:10.1016/0743-1066(90)90052-7.
- Freuder, Eugene; Alan Mackworth (ed.) (1994). *Constraint-based reasoning*. MIT Press.
- Frühwirth, Thom; Slim Abdennadher (2003). *Essentials of constraint programming*. Springer. ISBN 3-540-67623-6.
- Jaffar, Joxan; Michael J. Maher (1994). "Constraint logic programming: a survey". *Journal of logic programming* **19/20**: 503–581. doi:10.1016/0743-1066(94)90033-7.
- Laurière, Jean-Louis (1978). "A Language and a Program for Stating and Solving Combinatorial Problems". *Artificial intelligence* **10** (1): 29–127. doi:10.1016/0004-3702(78)90029-2.
- Lecoutre, Christophe (2009). *Constraint Networks: Techniques and Algorithms* ^[8]. ISTE/Wiley. ISBN 978-1-84821-106-3.

- Marriot, Kim; Peter J. Stuckey (1998). *Programming with constraints: An introduction*. MIT Press. ISBN 0-262-13341-5.
- Rossi, Francesca; Peter van Beek, Toby Walsh (ed.) (2006). *Handbook of Constraint Programming*, ^[9]. Elsevier. ISBN 978-0-444-52726-4 0-444-52726-5.
- Tsang, Edward (1993). *Foundations of Constraint Satisfaction* ^[10]. Academic Press. ISBN 0-12-701610-4.
- Van Hentenryck, Pascal (1989). *Constraint Satisfaction in Logic Programming*. MIT Press. ISBN 0-262-08181-4.

External links

- CSP Tutorial ^[11]

References

- [1] <http://jacop.osolpro.com/>
- [2] <http://www.koalog.com/>
- [3] <http://www.logilab.org/projects/constraint>
- [4] <http://minion.sourceforge.net/>
- [5] <http://www.bracil.net/CSP/cacp/cacpdemo.html>
- [6] <http://www.bracil.net/CSP/cacp/>
- [7] <http://www.ics.uci.edu/~dechter/books/index.html>
- [8] <http://www.iste.co.uk/index.php?f=a&ACTION=View&id=250>
- [9] http://www.elsevier.com/wps/find/bookdescription.cws_home/708863/description#description
- [10] <http://www.bracil.net/edward/FCS.html>
- [11] <http://4c.ucc.ie/web/outreach/tutorial.html>

Rewriting

In mathematics, computer science and logic, **rewriting** covers a wide range of (potentially non-deterministic) methods of replacing subterms of a formula with other terms. What is considered are **rewriting systems** (also known as **rewrite systems** or **reduction systems**). In their most basic form, they consist of a set of objects, plus relations on how to transform those objects.

Rewriting can be non-deterministic. One rule to rewrite a term could be applied in many different ways to that term, or more than one rule could be applicable. Rewriting systems then do not provide an algorithm for changing one term to another, but a set of possible rule applications. When combined with an appropriate algorithm, however, rewrite systems can be viewed as computer programs, and several declarative programming languages are based on term rewriting.

Intuitive examples

Logic

In logic, the procedure for determining the conjunctive normal form (CNF) of a formula can be conveniently written as a rewriting system. The rules of such a system would be:

- $\neg\neg A \rightarrow A$ (double negative elimination)
 - $\neg(A \wedge B) \rightarrow \neg A \vee \neg B$ (De Morgan's laws)
 - $\neg(A \vee B) \rightarrow \neg A \wedge \neg B$
 - $(A \wedge B) \vee C \rightarrow (A \vee C) \wedge (B \vee C)$ (Distributivity)
 - $A \vee (B \wedge C) \rightarrow (A \vee B) \wedge (A \vee C),$
-

where the arrow (\rightarrow) indicates that the left side of the rule can be rewritten to the right side (so it does not denote logical implication).

Abstract rewriting systems

From the above examples, it's clear that we can think of rewriting systems in an abstract manner. We need to specify a set of objects and the rules that can be applied to transform them. The most general (unidimensional) setting of this notion is called an **abstract reduction system**, (abbreviated **ARS**), although more recently authors use **abstract rewriting system** as well.^[1] (The preference for the word "reduction" here instead of "rewriting" constitutes a departure from the uniform use of "rewriting" in the names of systems that are particularizations of ARS. Because the word "reduction" does not appear in the names of more specialized systems, in older texts **reduction system** is a synonym for ARS).^[2]

An ARS is simply a set A , whose elements are usually called objects, together with a binary relation on A , traditionally denoted by \rightarrow , and called the **reduction relation**, **rewrite relation**^[3] or just **reduction**.^[2] This (entrenched) terminology using "reduction" is a little misleading, because the relation is not necessarily reducing some measure of the objects; this will become more apparent when we discuss string rewriting systems further in this article.

Example 1. Suppose the set of objects is $T = \{a, b, c\}$ and the binary relation is given by the rules $a \rightarrow b$, $b \rightarrow a$, $a \rightarrow c$, and $b \rightarrow c$. Observe that these rules can be applied to both a and b in any fashion to get the term c . Such a property is clearly an important one. Note also, that c is, in a sense, a "simplest" term in the system, since nothing can be applied to c to transform it any further. This example leads us to define some important notions in the general setting of an ARS. First we need some basic notions and notations.^[4]

- \rightarrow^* is the transitive closure of $\rightarrow \cup =$, where $=$ is the identity relation, i.e. \rightarrow^* is the smallest preorder (reflexive and transitive relation) containing \rightarrow . It is also called the reflexive transitive closure of \rightarrow .
- \leftrightarrow is $\rightarrow \cup \rightarrow^{-1}$, that is the union of the relation \rightarrow with its inverse relation, also known as the symmetric closure of \rightarrow .
- \leftrightarrow^* is the transitive closure of $\leftrightarrow \cup =$, that is \leftrightarrow^* is the smallest equivalence relation containing \rightarrow . It is also known as the reflexive transitive symmetric closure of \rightarrow .

Normal forms, joinability and the word problem

An object x in A is called **reducible** if there exist some other y in A and $x \rightarrow y$; otherwise it is called **irreducible** or a **normal form**. An object y is called a normal form of x if $x \rightarrow^* y$, and y is irreducible. If x has a *unique* normal form, then this is usually denoted with $x \downarrow$. In example 1 above, c is a normal form, and $c = a \downarrow = b \downarrow$. If every object has at least one normal form, the ARS is called **normalizing**.

A related, but weaker notion than the existence of normal forms is that of two objects being **joinable**: x and y are said joinable if there exists some z with the property that $x \rightarrow^* z \leftarrow^* y$. From this definition, it's apparent one may define the joinability relation as $\rightarrow^* \circ \leftarrow^*$, where \circ is the composition of relations. Joinability is usually denoted, somewhat confusingly, also with \downarrow , but in this notation the down arrow is a binary relation, i.e. we write $x \downarrow y$ if x and y are joinable.

One of the important problems that may be formulated in an ARS is the **word problem**: given x and y are they equivalent under \leftrightarrow^* ? This is a very general setting for formulating the word problem for the presentation of an algebraic structure. For instance, the word problem for groups is a particular case of an ARS word problem. Central to an "easy" solution for the word problem is the existence of unique normal forms: in this case if two objects have the same normal form, then they are equivalent under \leftrightarrow^* . The word problem for an ARS is undecidable in general.

The Church-Rosser property and confluence

An ARS is said to possess the **Church-Rosser property** if and only if $x \xleftrightarrow{*} y$ implies $x \downarrow y$. In words, the Church-Rosser property means that the reflexive transitive symmetric closure is contained in the joinability relation. Alonzo Church and J. Barkley Rosser proved in 1936 that lambda calculus has this property;^[5] hence the name of the property.^[6] (The fact that lambda calculus has this property is also known as the Church-Rosser theorem.) In an ARS with the Church-Rosser property the word problem may be reduced to the search for a common successor. In a Church-Rosser system, an object has *at most one* normal form; that is the normal form of an object is unique if it exists, but it may well not exist.

Several different properties are equivalent to the Church-Rosser property, but may be simpler to check in any particular setting. In particular, *confluence* is equivalent to Church-Rosser. The notion of confluence can be defined for individual elements, something that's not possible for Church-Rosser. An ARS (A, \rightarrow) is said:

- **confluent** if and only if for all w, x , and y in A , $x \xleftarrow{*} w \xrightarrow{*} y$ implies $x \downarrow y$. Roughly speaking, confluence says that no matter how two paths diverge from a common ancestor (w), the paths are joining at *some* common successor. This notion may be refined as property of a particular object w , and the system called confluent if all its elements are confluent.
- **locally confluent** if and only if for all w, x , and y in A , $x \leftarrow w \rightarrow y$ implies $x \downarrow y$. This property is sometimes called **weak confluence**.

Theorem. For an ARS the following conditions are equivalent: (i) it has the Church-Rosser property, (ii) it is confluent.^[7]

Corollary.^[8] In a confluent ARS if $x \xleftrightarrow{*} y$ then

- If both x and y are normal forms, then $x = y$.
- If y is a normal form, then $x \xrightarrow{*} y$

Because of these equivalences, a fair bit of variation in definitions is encountered in the literature. For instance, in Bezem et al 2003 the Church-Rosser property and confluence are defined to be synonymous and identical to the definition of confluence presented here; Church-Rosser as defined here remains unnamed, but is given as an equivalent property; this departure from other texts is deliberate.^[9] Because of the above corollary, one may define a normal form y of x as an irreducible y with the property that $x \xleftrightarrow{*} y$. This definition, found in Book and Otto, is equivalent to common one given here in a confluent system, but it is more inclusive in a non-confluent ARS.

Local confluence on the other hand is not equivalent with the other notions of confluence given in this section, but it is strictly weaker than confluence.

Termination and convergence

An abstract rewriting system is said to be **terminating** or **noetherian** if there is no infinite chain $x_0 \rightarrow x_1 \rightarrow x_2 \rightarrow \dots$. In a terminating ARS, every object has at least one normal form, thus it is normalizing. The converse is not true. In example 1 for instance, there is an infinite rewriting chain, namely $a \rightarrow b \rightarrow a \rightarrow b \rightarrow \dots$, even though the system is normalizing. A confluent and terminating ARS is called **convergent**. In a convergent ARS, every object has a unique normal form. But it is sufficient for the system to be confluent and normalizing for a unique normal to exist for every element, as seen in example 1.

Theorem (Newman's Lemma): A terminating ARS is confluent if and only if it is locally confluent.

String rewriting systems

A **string rewriting system** (SRS), also known as **semi-Thue system**, exploits the free monoid structure of the strings (words) over an alphabet to extend a rewriting relation, R to *all* strings in the alphabet that contain left- and respectively right-hand sides of some rules as substrings. Formally a semi-Thue systems is a tuple (Σ, R) where Σ is a (usually finite) alphabet, and R is a binary relation between some (fixed) strings in the alphabet, called **rewrite rules**. The **one-step rewriting relation** \rightarrow_R induced by R on Σ^* is defined as: for any strings s , and t in Σ^* $s \rightarrow_R t$ if and only if there exist x, y, u, v in Σ^* such that $s = xuy$, $t = xvy$, and $u R v$. Since \rightarrow_R is a relation on Σ^* , the pair $(\Sigma^*, \rightarrow_R)$ fits the definition of an abstract rewriting system. Obviously R is subset of \rightarrow_R . If the relation R is symmetric, then the system is called a **Thue system**.

In a SRS, the **reduction relation** $\xrightarrow{*}_R$ is compatible with the monoid operation, meaning that $x \xrightarrow{*}_R y$ implies $uxv \xrightarrow{*}_R u y v$ for all strings x, y, u, v in Σ^* . Similarly, the reflexive transitive symmetric closure of \rightarrow_R , denoted \leftrightarrow_R^* , is a congruence, meaning it is an equivalence relation (by definition) and it is also compatible with string concatenation. The relation \leftrightarrow_R^* is called the **Thue congruence** generated by R . In a Thue system, i.e. if R is symmetric, the rewrite relation $\xrightarrow{*}_R$ coincides with the Thue congruence \leftrightarrow_R^* . The notion of a semi-Thue system essentially coincides with the presentation of a monoid. Since \leftrightarrow_R^* is a congruence, we can define the **factor monoid** $\mathcal{M}_R = \Sigma^* / \leftrightarrow_R^*$ of the free monoid Σ^* by the Thue congruence in the usual manner. If a monoid \mathcal{M} is isomorphic with \mathcal{M}_R , then the semi-Thue system (Σ, R) is called a monoid presentation of \mathcal{M} .

We immediately get some very useful connections with other areas of algebra. For example, the alphabet $\{a, b\}$ with the rules $\{ab \rightarrow \varepsilon, ba \rightarrow \varepsilon\}$, where ε is the empty string, is a presentation of the free group on one generator. If instead the rules are just $\{ab \rightarrow \varepsilon\}$, then we obtain a presentation of the bicyclic monoid. Thus semi-Thue systems constitute a natural framework for solving the word problem for monoids and groups. In fact, every monoid has a presentation of the form (Σ, R) , i.e. it may be always be presented by a semi-Thue system, possibly over an infinite alphabet.

The word problem for a semi-Thue system is undecidable in general; this result is sometimes know as the **Post-Markov theorem**.^[10]

Term rewriting systems

A generalization of term rewrite systems are graph rewrite systems, operating on graphs instead of (ground-) terms / their corresponding tree representation.

Trace rewriting systems

Trace theory provides a means for discussing multiprocessing in more formal terms, such as via the trace monoid and the history monoid. Rewriting can be performed in trace systems as well.

Philosophy

Rewriting systems can be seen as programs that infer end-effects from a list of cause-effect relationships. In this way, rewriting systems can be considered to be automated causality provers.

Properties of rewriting systems

Observe that in both of the above rewriting systems, it is possible to get terms rewritten to a "simplest" term, where this term cannot be modified any further from the rules in the rewriting system. Terms which cannot be written any further are called *normal forms*. The potential existence or uniqueness of normal forms can be used to classify and describe certain rewriting systems. There are rewriting systems which do not have normal forms: a very simple one is the rewriting system on two terms a and b with $a \rightarrow b, b \rightarrow a$.

The property exhibited above where terms can be rewritten regardless of the choice of rewriting rule to obtain the same normal form is known as *confluence*. The property of confluence is linked with the property of having unique normal forms.

See also

- Critical pair
- Knuth-Bendix completion algorithm
- L-system
- Pure a programming language based on term rewriting.
- Regulated rewriting
- Rho calculus

References

- Franz Baader and Tobias Nipkow (1999). *Term rewriting and all that* ^[11]. Cambridge University Press. ISBN 0-521-77920-0. 316 pages. A textbook suitable for undergraduates.
- Marc Bezem, J. W. Klop, Roel de Vrijer ("Terese"), *Term rewriting systems*, Cambridge University Press, 2003, ISBN 0521391156. This is the most recent comprehensive monograph. It uses however a fair deal of non-yet-standard notations and definitions. For instance the Church-Rosser property is defined to be identical with confluence.
- Ronald V. Book and Friedrich Otto, *String-rewriting Systems*, Springer (1993).
- Nachum Dershowitz and Jean-Pierre Jouannaud *Rewrite Systems* ^[12], Chapter 6 in Jan van Leeuwen (Ed.), *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics.*, Elsevier and MIT Press, 1990, ISBN 0-444-88074-7, pp. 243–320. The preprint of this chapter is freely available from the authors, but it misses the figures.

External links

- The Rewriting Home Page ^[13]

References

- [1] Bezem et al, p. 7,
- [2] Book and Otto, p. 10
- [3] Bezem et al, p. 7
- [4] Baader and Nipkow, pp. 8-9
- [5] Alonzo Church and J. Barkley Rosser. Some properties of conversion. Trans. AMS, 39:472-482, 1936
- [6] Baader and Nipkow, p. 9
- [7] Baader and Nipkow, p. 11
- [8] Baader and Nipkow, p. 12
- [9] Bezem et al, p.11
- [10] Martin Davis et al. 1994, p. 178
- [11] <http://books.google.com/books?id=N7BvXVUCQk8C&dq>
- [12] <http://citeseer.ist.psu.edu/dershowitz90rewrite.html>

[13] <http://rewriting.loria.fr/>

Maude system

The **Maude system** is an implementation of rewriting logic developed at SRI International. It is similar in its general approach to Joseph Goguen's OBJ3 implementation of equational logic, but based on rewriting logic rather than order-sorted equational logic, and with a heavy emphasis on powerful metaprogramming based on reflection.

Maude - the basics

Maude modules (rewrite theories) consists of a term-language plus sets of equations and rewrite-rules. Terms in a rewrite theory are constructed using operators (functions taking 0 or more arguments of some **sort**, which return a term of a specific **sort**). Operators taking 0 arguments are considered constants, and we construct our term-language by these simple constructs.

NOTE: If you want to test these examples on your own, you should start Maude with the option **-no-prelude** which lets Maude know that none of its basic modules are included (like Maude's own NAT module which will cause a conflict).

Example 1

```
fmod NAT is
  sort Nat .

  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
endfm
```

This rewrite theory specifies all the natural numbers. First we introduce the **sort**, saying that there exists a sort called Nat (short for natural numbers), and below is the specification of how these terms are constructed. The operator **s** in Example 1 is the successor function representing the next natural number in the sequence of natural numbers i.e. $s(N) := N + 1$. $s(0)$ is meant to represent the natural number 1 and so on. **0** is a constant, it takes no input parameter(s) but returns a **Nat**.

Example 2

```
fmod NAT is
  sort Nat .

  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  op _+_ : Nat Nat -> Nat .

  vars N M : Nat .

  eq 0 + N = N .
  eq s(N) + M = s (N + M) .

endfm
```

In Example 2 we introduce the $+$ sign meant to represent addition of natural numbers. Its definition is almost identical to the previous one, with input and output **sorts** but there is a difference, its operator has underscores on each side. Maude lets the user specify whether or not operators are infix, postfix or prefix (default), this is done using underscores as place fillers for the input terms. So the $+$ operator takes its input on each side making it an infix operator. Unlike our previous operator s which takes its input terms after the operator (prefix).

```
op + : Nat Nat -> Nat .
*** is the same as

op +__ : Nat Nat -> Nat . *** two underscores
```

The three stars are Maude's rest-of-line-comments and lets the parser know that it can ignore the rest of the line (not part of the program), with parenthesis we get section comments:

```
*** rest of line is ignored by Maude
*** (
section
is
ignored by Maude
)
```

The extended **Nat** module also holds two variables and two sets of equations.

```
vars N M : Nat .

eq 0 + N = N .
eq s(N) + M = s (N + M) .
```

When Maude "executes", it rewrites terms according to our specifications. And this is done with the statement

```
reduce in <some module> : <some term> .
```

or the shorter form

```
red <some term> .
```

For this last statement to be used it is important that nothing is ambiguous. A small example from our rewrite theory representing the natural numbers:

```
reduce in NAT : s(0) + s(0) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result Nat: s(s(0))
```

Using the two equations in the rewrite theory **NAT** Maude was able to reduce our term into the desired term, the representation of the number two i.e. the second successor of **0**. At that point no other application of the two equations were possible, so Maude halts. Maude rewrites terms according to the equations whenever there is a **match** between the **closed terms** that we try to rewrite (or reduce) and the **left hand side** of an equation in our equation-set. A match in this context is a substitution of the variables in the left hand side of an equation which leaves it identical to the term we try to rewrite/reduce.

To illustrate it further we can look at the left hand side of the equations as Maude executes, reducing/rewriting the term.

```
eq s(N) + M = s (N + M) .
```

can be applied to the term:

```
s(0) + s(0)
```

since the substitution:

```
N => 0
```

```
M => s(0)
```

```
s(N) + M N => 0, M => s(0) == s(0) + s(0)
```

makes them identical, and therefore it can be reduced/rewritten using this equation. After this equation has been applied to the term, we are left with the term:

```
s(0 + s(0))
```

If we take a close look at that term we will see that it has a fitting substitution with matches the first equation, at least parts of the term matches the first equation:

```
eq 0 + N = N .
```

substitution:

```
N => s(0)
```

```
s(0 + N) N => s(0) == s(0 + s(0))
```

```
0 + s(0) - matches first equation and is rewritten
```

As we can see the second substitution and rewrite step rewrites the an inner-term (the whole term does not match any of the equations but the inner term does). Then we end up with our resulting term $s(s(0))$, and it can seem like a lot of hassle to add $1 + 1$, but hopefully you will soon see the strength of this approach.

Another thing worth mentioning is that reduction/rewriting up to this point has taken something very important for granted, which is:

- Reduction/Rewriting **terminates**
- Reduction/Rewriting is **confluent** (applying the equations in any order will eventually lead to the same resulting term)

This cannot be taken for granted, and for a rewrite theory to be sane, we have to ensure that equational application is confluent and terminating. To prove that term-rewriting terminates is not possible in any instance, as we know from the halting problem. To be able to prove that term-rewriting (with regards to the equations) terminates, one can usually create some mapping between terms and the natural numbers, and show that application of the equations reduces the associated value of the term. Induction then proves that it is a halting process since the event of finding smaller natural numbers is a halting process. Naturally equation sets that can cause a term-rewrite to contain cycles will not be terminating. To prove confluence is another important aspect since a rewrite theory lacking this ability will be rather flawed. To prove that the equation set is confluent one has to prove that **any** application of the equations to any belonging term will lead to the same resulting term (termination is a prerequisite). Details for how to prove termination or confluence will not be given here, it just had to be mentioned, since this is where equations and rewrite-rules differ, which is the next topic of this short overview.

Rewrite Rules

Up to this point we have been talking about rewriting and reduction as more or less the same thing, our first rewrite theory had no rewrite rules, still we rewrote terms, so it is time to illustrate what rewrite rules are, and how they differ from the equations we have seen so far (they do not differ much from equations naturally since we talk about the two concepts almost interchangeably).

The module presented so far **NAT** which represented the natural numbers and addition on its elements, is considered a functional module/rewrite theory, since it contains no rewrite rules. Such modules are often encapsulated with a **fmod** and **endfm** (instead of **mod endm**) to illustrate this fact. A functional module and its set of equations must be confluent and terminating since they build up the part of a rewrite theory that always should compute the same result, this will be clear once the rewrite rules come into play.

Example 3

```
mod PERSON is

including NAT . *** our previous module

sort Person .
sort State .

op married : -> State [ctor] .
op divorced : -> State [ctor] .
op single : -> State [ctor] .
op engaged : -> State [ctor] .
op dead : -> State [ctor] .

op person : State Nat -> Person [ctor] .

var N : Nat .
var S : State .

rl [birthday] :

person (S, N) => person (S, N + s(0)) .

rl [get-engaged] :

person (single, N) => person (engaged, N) .

rl [get-married] :

person (engaged, N) => person (married, N) .

rl [get-divorced] :

person (married, N) => person (divorced, N) .

rl [las-vegas] :
```

```

person (S, N) => person (married, N) .

rl [die] :

person (S, N) => person (dead, N) .

endm

```

This module introduces two new **sorts**, and a set of rewrite rules. We also include our previous module, to illustrate how equations and rewrite rules differ. The rewrite rules is thought of as a set of legal state changes, so while equations hold the same meaning on both sides of the equality sign, rewrite rules do not (rewrite rules use a `=>` token instead of an equality sign). You are still the same person after your married (this is open for debate), but something has changed, your marital status at least. So this is illustrated by a rewrite rule, not an equation. Rewrite rules do **not** have to be **confluent** and **terminating** so it does matter a great deal what rules are chosen to rewrite the term. The rules are applied at "random" by the Maude system, meaning that you can not be sure that one rule is applied before another rule and so on. **If** an equation can be applied to the term, it will **always** be applied **before** any rewrite rule.

A small example is in order:

Example 4

```

rewrite [3] in PERSON : person (single, 0) .

rewrites: 4 in 0ms cpu (0ms real) (~ rewrites/second)

result Person: person (married, s(0))

```

Here we tell the Maude system to rewrite our input term according to the rewrite theory, and we tell it to stop after 3 rewrite steps (remember that rewrite rules do not have to be terminating or confluent so an upper bound is not a bad idea), we just want see what sort of state we end up in after randomly choosing 3 **matching** rules. And as you can see the state this person ends up in might look a bit strange. (When you're married at the age of one you kind of stick out a bit in kindergarten I guess). It also says 4 rewrite steps, although we specifically stated an upper limit of 3 rewrite steps, this is because rewrite steps that are the result of applying equations does not count (they do not change the term, at least if the equations are sane). In this example one of the equations of the NAT module has been used to reduce the term $0 + s(0)$ to $s(0)$, which accounts for the 4'th rewrite step.

To make this rewrite theory a bit less morbid, we can alter some of our rewrite rules a bit, and make them **conditional** rewrite rules, which basically means they have to fulfill some criteria to be applied to the term (other than just matching the left hand side of the rewrite rule).

```

crl [las-vegas] :

person (S, N) => person (married, N) if (S /= married) /\ (S /= dead) .

crl [die] :

person (S, N) => person (dead, N) if (S /= dead) .

```

It seems reasonable that you cannot die once you're dead, and you cannot marry as long as you are married. The leaning toothpicks (`/\`) are supposed to resemble a logical **AND** which means that both criteria have to be fulfilled to be able to apply the rule (apart from term matching). **Equations** can also be made conditional in a similar manner.

Why Rewriting Logic? Why Maude?

As you probably can tell Maude sets out to solve a different set of problems than ordinary imperative languages like C, Java or Perl. It is a formal reasoning tool, which can help us verify that things are "as they should", and show us why they are not if this is the case. In other words Maude lets us define formally what we mean by some concept in a very abstract manner (not concerning ourselves with how the structure is internally represented and so on), but we can describe what is thought to be the equal concerning our theory (**equations**) and what state changes it can go through (**rewrite rules**). This is extremely useful to validate security protocols and critical code. The Maude system has proved flaws in cryptography protocols by just specifying what the system can do (in a manner similar to the PERSON rewrite theory), and by looking for unwanted situations (states or terms that should not be possible to reach) the protocol can be showed to contain bugs, not programming bugs but situations happen that are hard to predict just by walking down the "happy path" as most developers do.

We can use Maude's built-in search to look for unwanted states, or it can be used to show that no such states can be reached.

A small example from our PERSON module once again.

```
search [1] in PERSON : person (single, 1) =>1 person (married, 2) .
```

```
No solution.
```

Here the Natural numbers have a more familiar look (the basic Maude modules prelude.maunder has been loaded, that can be done with the command "**in prelude**", or 1 can be replaced by s(0) and 2 by s(s(0)) if you don't want to load the default Maude-modules), naturally Maude has built-in support for ordinary structures like integers and floats and so on. The natural numbers are still members of the built-in **sort** Nat. We state that we want to search for a transition using one rewrite rule (=>1), which rewrites the first term into the other. The result of the investigation is not shocking but still, sometimes proving the obvious is all we can do. If we let Maude use more than one step we should see a different result:

```
search [1] in PERSON : person (single, 1) =>+ person (married, 2) .
```

```
Solution 1 (state 7)
```

```
states: 8 rewrites: 14 in 0ms cpu (0ms real) (~ rewrites/second)
```

To see what led us in that direction we can use the built in command **show path** which lets us know what rule applications led us to the resulting term. The token (=>+) means one or more rule application.

```
show path 7 .
```

```
state 0, Person: person (single, 1)
```

```
===[rl person(S, N) => person(S, N + 1) [label 'birthday']] .]
```

```
==>
```

```
state 1, Person: person (single, 2)
```

```
===[crl person(S, N) => person(married, N) if S /= married = true /\ S /= dead = true [label las-vegas]] .]
```

```
==>
```

```
state 7, Person: person (married, 2)
```

As we can see the rule application "birthday" followed by "las-vegas" got us where we wanted. Since all rewrite theories or modules with many possible rule applications will generate a huge tree of possible states to search for with the **search** command, this approach is not always the solution. We also have the ability to control what rule applications should be attempted at each step using meta-programming, due to the reflective property or rewriting logic. This subject of meta-programming and reflection requires a great deal of explanation for it to be simple, so that will be left for another section.

Maude is free software with great performance, and well written tutorials are available online (don't let this short howto scare you away).

References

- Clavel, Durán, Eker, Lincoln, Martí-Oliet, Meseguer and Quesada, 1998. *Maude as a Metalanguage* ^[1], in Proc. 2nd International Workshop on Rewriting Logic and its Applications, Electronic Notes in Theoretical Computer Science 15, Elsevier.
- Martí-Oliet and José Meseguer, 2002. *Rewriting Logic: Roadmap and Bibliography* ^[2]. Theoretical Computer Science 285(2):121-154.
- Martí-Oliet and José Meseguer, 1993-2000. *Rewriting Logic as a Logical and Semantic Framework* ^[3] [4]. Electronic Notes in Theoretical Computer Science 4, Elsevier.

External links

- Maude homepage ^[5] at University of Illinois at Urbana-Champaign;
- The Real-Time Maude Tool homepage ^[6] developed by Peter Csaba Ölveczky;
- An introduction to Maude ^[7] by Neal Harman, Swansea University (errata ^[8])
- The Policy And GOal based Distributed Architecture ^[9] written in Maude by SRI International.
- Maude for Windows ^[10], the Maude installer for Windows, and Maude Development Tools ^[11], the Maude Eclipse plugin developed by the MOMENT project ^[12] at Technical University of Valencia ^[13] (Spain).

References

- [1] http://maude.cs.uiuc.edu/papers/abstract/CDELMMQmetalanguage_1998.html
- [2] http://maude.cs.uiuc.edu/papers/abstract/MMroadmap_2001.html
- [3] <http://maude.cs.uiuc.edu/papers/abstract/tcs4012.html>
- [4] http://maude.cs.uiuc.edu/papers/abstract/MMlogframework_1993.html
- [5] <http://maude.cs.uiuc.edu/>
- [6] <http://heim.ifi.uio.no/~peterol/RealTimeMaude/>
- [7] <http://www.cs.swan.ac.uk/~csneal/MaudeCourse/index.html>
- [8] <http://www.cs.swan.ac.uk/~csmona/cs213/errata.txt>
- [9] <http://pagoda.csl.sri.com/>
- [10] <http://moment.dsic.upv.es/mfw/>
- [11] <http://moment.dsic.upv.es/mdt/>
- [12] <http://moment.dsic.upv.es>
- [13] <http://www.upv.es>

Resolution (logic)

In mathematical logic and automated theorem proving, **resolution** is a rule of inference leading to a refutation theorem-proving technique for sentences in propositional logic and first-order logic. In other words, iteratively applying the resolution rule in a suitable way allows for telling whether a propositional formula is satisfiable and for proving that a first-order formula is unsatisfiable; this method may prove the satisfiability of a first-order satisfiable formula, but not always, as it is the case for all methods for first-order logic. Resolution was introduced by John Alan Robinson in 1965.

Resolution in propositional logic

Resolution rule

The **resolution rule** in propositional logic is a single valid inference rule that produces a new clause implied by two clauses containing complementary literals. A literal is a propositional variable or the negation of a propositional variable. Two literals are said to be complements if one is the negation of the other (in the following, a_i is taken to be the complement to b_j). The resulting clause contains all the literals that do not have complements. Formally:

$$\frac{a_1 \vee \dots \vee a_i \vee \dots \vee a_n, \quad b_1 \vee \dots \vee b_j \vee \dots \vee b_m}{a_1 \vee \dots \vee a_{i-1} \vee a_{i+1} \vee \dots \vee a_n \vee b_1 \vee \dots \vee b_{j-1} \vee b_{j+1} \vee \dots \vee b_m}$$

where

all a s and b s are literals,

a_i is the complement to b_j , and

the dividing line stands for entails

The clause produced by the resolution rule is called the *resolvent* of the two input clauses.

When the two clauses contain more than one pair of complementary literals, the resolution rule can be applied (independently) for each such pair. However, only the pair of literals that are resolved upon can be removed: all other pairs of literals remain in the resolvent clause. Note that resolving any two clauses that can be resolved over more than one variable always results in a tautology.

Modus ponens can be seen as a special case of resolution of a one-literal clause and a two-literal clause.

A resolution technique

When coupled with a complete search algorithm, the resolution rule yields a sound and complete algorithm for deciding the *satisfiability* of a propositional formula, and, by extension, the validity of a sentence under a set of axioms.

This resolution technique uses proof by contradiction and is based on the fact that any sentence in propositional logic can be transformed into an equivalent sentence in conjunctive normal form. The steps are as follows.

- All sentences in the knowledge base and the *negation* of the sentence to be proved (the *conjecture*) are conjunctively connected.
- The resulting sentence is transformed into a conjunctive normal form with the conjuncts viewed as elements in a set, S , of clauses.
 - For example $(A_1 \vee A_2) \wedge (B_1 \vee B_2 \vee B_3) \wedge (C_1)$ gives rise to the set $S = \{A_1 \vee A_2, B_1 \vee B_2 \vee B_3, C_1\}$.
- The resolution rule is applied to all possible pairs of clauses that contain complementary literals. After each application of the resolution rule, the resulting sentence is simplified by removing repeated literals. If the sentence contains complementary literals, it is discarded (as a tautology). If not, and if it is not yet present in the clause set

S , it is added to S , and is considered for further resolution inferences.

- If after applying a resolution rule the *empty clause* is derived, the original formula is unsatisfiable (or *contradictory*), and hence it can be concluded that the initial conjecture follows from the axioms.
- If, on the other hand, the empty clause cannot be derived, and the resolution rule cannot be applied to derive any more new clauses, the conjecture is not a theorem of the original knowledge base.

One instance of this algorithm is the original Davis–Putnam algorithm that was later refined into the DPLL algorithm that removed the need for explicit representation of the resolvents.

This description of the resolution technique uses a set S as the underlying data-structure to represent resolution derivations. Lists, Trees and Directed Acyclic Graphs are other possible and common alternatives. Tree representations are more faithful to the fact that the resolution rule is binary. Together with a sequent notation for clauses, a tree representation also makes it clear to see how the resolution rule is related to a special case of the cut-rule, restricted to atomic cut-formulas. However, tree representations are not as compact as set or list representations, because they explicitly show redundant subderivations of clauses that are used more than once in the derivation of the empty clause. Graph representations can be as compact in the number of clauses as list representations and they also store structural information regarding which clauses were resolved to derive each resolvent.

A simple example

$$\frac{a \vee b, \quad \neg a \vee c}{b \vee c}$$

In English: if a or b is true, and a is false or c is true, then either b or c is true.

If a is true, then for the second premise to hold, c must be true. If a is false, then for the first premise to hold, b must be true.

So regardless of a , if both premises hold, then b or c is true.

Resolution in first order logic

In first order logic, resolution condenses the traditional syllogisms of logical inference down to a single rule.

To understand how resolution works, consider the following example syllogism of term logic:

All Greeks are Europeans.

Homer is a Greek.

Therefore, Homer is a European.

Or, more generally:

$$\forall x. P(x) \Rightarrow Q(x)$$

$$P(a)$$

$$\text{Therefore, } Q(a)$$

To recast the reasoning using the resolution technique, first the clauses must be converted to conjunctive normal form. In this form, all quantification becomes implicit: universal quantifiers on variables (X , Y , ...) are simply omitted as understood, while existentially-quantified variables are replaced by Skolem functions.

$$\neg P(x) \vee Q(x)$$

$$P(a)$$

$$\text{Therefore, } Q(a)$$

So the question is, how does the resolution technique derive the last clause from the first two? The rule is simple:

- Find two clauses containing the same predicate, where it is negated in one clause but not in the other.

- Perform a unification on the two predicates. (If the unification fails, you made a bad choice of predicates. Go back to the previous step and try again.)
- If any unbound variables which were bound in the unified predicates also occur in other predicates in the two clauses, replace them with their bound values (terms) there as well.
- Discard the unified predicates, and combine the remaining ones from the two clauses into a new clause, also joined by the " \vee " operator.

To apply this rule to the above example, we find the predicate P occurs in negated form

$$\neg P(X)$$

in the first clause, and in non-negated form

$$P(a)$$

in the second clause. X is an unbound variable, while a is a bound value (atom). Unifying the two produces the substitution

$$X \mapsto a$$

Discarding the unified predicates, and applying this substitution to the remaining predicates (just $Q(X)$, in this case), produces the conclusion:

$$Q(a)$$

For another example, consider the syllogistic form

All Cretans are islanders.

All islanders are liars.

Therefore all Cretans are liars.

Or more generally,

$\forall X P(X)$ implies $Q(X)$

$\forall X Q(X)$ implies $R(X)$

Therefore, $\forall X P(X)$ implies $R(X)$

In CNF, the antecedents become:

$$\neg P(X) \vee Q(X)$$

$$\neg Q(Y) \vee R(Y)$$

(Note that the variable in the second clause was renamed to make it clear that variables in different clauses are distinct.)

Now, unifying $Q(X)$ in the first clause with $\neg Q(Y)$ in the second clause means that X and Y become the same variable anyway. Substituting this into the remaining clauses and combining them gives the conclusion:

$$\neg P(X) \vee R(X)$$

The resolution rule, as defined by Robinson, also incorporated factoring, which unifies two literals in the same clause, before or during the application of resolution as defined above. The resulting inference rule is refutation complete, in that a set of clauses is unsatisfiable if and only if there exists a derivation of the empty clause using resolution alone.

Implementations

- Carine
- Gandalf
- Otter
- Prover9
- SNARK
- SPASS
- Vampire

See also

- Conjunctive normal form
- Inverse resolution

References

- J. Alan Robinson (1965), A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM* (JACM), Volume 12, Issue 1, pp. 23–41.
- Leitsch, Alexander (1997). *The Resolution Calculus*. Springer-Verlag.
- Gallier, Jean H. (1986). *Logic for Computer Science: Foundations of Automatic Theorem Proving*^[1]. Harper & Row Publishers.

External links

- Alex Sakharov, "Resolution Principle^[2]" from MathWorld.
- Alex Sakharov, "Resolution^[3]" from MathWorld.
- Notes on computability and resolution^[4]

References

- [1] <http://www.cis.upenn.edu/~jean/gbooks/logic.html>
- [2] <http://mathworld.wolfram.com/ResolutionPrinciple.html>
- [3] <http://mathworld.wolfram.com/Resolution.html>
- [4] http://www.cs.uu.nl/docs/vakken/pv/resources/computational_prop_of_fol.pdf
-

Automated theorem proving

Automated theorem proving (ATP) or **automated deduction**, currently the most well-developed subfield of *automated reasoning* (AR), is the proving of mathematical theorems by a computer program.

Decidability of the problem

Depending on the underlying logic, the problem of deciding the validity of a formula varies from trivial to impossible. For the frequent case of propositional logic, the problem is decidable but NP-complete, and hence only exponential-time algorithms are believed to exist for general proof tasks. For a first order predicate calculus, with no ("proper") axioms, Gödel's completeness theorem states that the theorems (provable statements) are exactly the logically valid well-formed formulas, so identifying valid formulas is recursively enumerable: given unbounded resources, any valid formula can eventually be proven.

However, *invalid* formulas (those that are *not* entailed by a given theory), cannot always be recognized. In addition, a consistent formal theory that contains the first-order theory of the natural numbers (thus having certain "proper axioms"), by Gödel's incompleteness theorem, contains true statements which cannot be proven. In these cases, an automated theorem prover may fail to terminate while searching for a proof. Despite these theoretical limits, in practice, theorem provers can solve many hard problems, even in these undecidable logics.

Related problems

A simpler, but related, problem is proof verification, where an existing proof for a theorem is certified valid. For this, it is generally required that each individual proof step can be verified by a primitive recursive function or program, and hence the problem is always decidable.

Interactive theorem provers require a human user to give hints to the system. Depending on the degree of automation, the prover can essentially be reduced to a proof checker, with the user providing the proof in a formal way, or significant proof tasks can be performed automatically. Interactive provers are used for a variety of tasks, but even fully automatic systems have proven a number of interesting and hard theorems, including some that have eluded human mathematicians for a long time.^{[1] [2]} However, these successes are sporadic, and work on hard problems usually requires a proficient user.

Another distinction is sometimes drawn between theorem proving and other techniques, where a process is considered to be theorem proving if it consists of a traditional proof, starting with axioms and producing new inference steps using rules of inference. Other techniques would include model checking, which is equivalent to brute-force enumeration of many possible states (although the actual implementation of model checkers requires much cleverness, and does not simply reduce to brute force).

There are hybrid theorem proving systems which use model checking as an inference rule. There are also programs which were written to prove a particular theorem, with a (usually informal) proof that if the program finishes with a certain result, then the theorem is true. A good example of this was the machine-aided proof of the four color theorem, which was very controversial as the first claimed mathematical proof which was essentially impossible to verify by humans due to the enormous size of the program's calculation (such proofs are called non-surveyable proofs). Another example would be the proof that the game Connect Four is a win for the first player.

Industrial uses

Commercial use of automated theorem proving is mostly concentrated in integrated circuit design and verification. Since the Pentium FDIV bug, the complicated floating point units of modern microprocessors have been designed with extra scrutiny. In the latest processors from AMD, Intel, and others, automated theorem proving has been used to verify that division and other operations are correct.

First-order theorem proving

First-order theorem proving is one of the most mature subfields of automated theorem proving. The logic is expressive enough to allow the specification of arbitrary problems, often in a reasonably natural and intuitive way. On the other hand, it is still semi-decidable, and a number of sound and complete calculi have been developed, enabling *fully* automated systems. More expressive logics, such as higher order and modal logics, allow the convenient expression of a wider range of problems than first order logic, but theorem proving for these logics is less well developed. The quality of implemented system has benefited from the existence of a large library of standard benchmark examples — the Thousands of Problems for Theorem Provers (TPTP) Problem Library^[3] — as well as from the CADE ATP System Competition (CASC), a yearly competition of first-order systems for many important classes of first-order problems.

Some important systems (all have won at least one CASC competition division) are listed below.

- E is a high-performance prover for full first-order logic, but built on a purely equational calculus, developed primarily in the automated reasoning group of Technical University of Munich.
- Otter, developed at the Argonne National Laboratory, is the first widely used high-performance theorem prover. It is based on first-order resolution and paramodulation. Otter has since been replaced by Prover9, which is paired with Mace4.
- SETHEO is a high-performance system based on the goal-directed model elimination calculus. It is developed in the automated reasoning group of Technical University of Munich. E and SETHEO have been combined (with other systems) in the composite theorem prover E-SETHEO.
- Vampire is developed and implemented at Manchester University^[4] by Andrei Voronkov^[5], formerly together with Alexandre Riazanov^[6]. It has won the "world cup for theorem provers" (the CADE ATP System Competition) in the most prestigious CNF (MIX) division for eight years (1999, 2001–2007).
- Waldmeister is a specialized system for unit-equational first-order logic. It has won the CASC UEQ division for the last ten years (1997–2006).

Deontic theorem proving

Deontic logic concerns normative propositions, such as those used in law, engineering specifications, and computer programs. In other words, propositions that are translations of commands or "ought" or "must (not)" statements in ordinary language. The deontic character of such logic requires formalism that extends the first-order predicate calculus. Representative of this is the tool KED.^[7]

Popular techniques

- First-order resolution with unification
 - Lean theorem proving
 - Model elimination
 - Method of analytic tableaux
 - Superposition and term rewriting
 - Model checking
 - Mathematical induction
-

- Binary decision diagrams
- DPLL
- Higher-order unification

Available implementations

See also: Category:Theorem proving software systems

Free software

- | | | | |
|------------------|-------------------------|----------------------|----------------------|
| • ACL2 | • iProver | • LoTREC | • SAD ^[8] |
| • Alt-Ergo | • IsaPlanner | • MetaPRL | • SNARK |
| • Automath | • Jape | • NuPRL | • SPASS |
| • CVC | • KED | • Otter | • Tau |
| • E | • KeY | • Paradox | • Theorema |
| • EQP | • KeYmaera | • PhoX | • Theorem Checker |
| • Gandalf | • LCF | • Prover9 / Mace4 | |
| • Gödel-machines | • Leo II ^[9] | • PVS | |

Proprietary software including Share-alike Non-commercial

- | | |
|--|---|
| • Acumen RuleManager (commercial product) | • Simplify |
| • Alligator | • SPARK (programming language) |
| • CARINE | • Spear modular arithmetic theorem prover |
| • KIV | • Theorem Proving System (TPS) |
| • Prover Plug-In (commercial proof engine product) | • Twelf |
| • ProverBox | • Vampire/Vampyre |
| • ResearchCyc | • Waldmeister |

Notable people

- Leo Bachmair, co-developer of the superposition calculus.
- Woody Bledsoe, artificial intelligence pioneer.
- Robert S. Boyer, co-author of the Boyer-Moore theorem prover, co-recipient of the Herbrand Award 1999.
- Alan Bundy, University of Edinburgh, meta-level reasoning for guiding inductive proof, proof planning and recipient of 2007 IJCAI Award for Research Excellence, Herbrand Award, and 2003 Donald E. Walker Distinguished Service Award.
- William McCune ^[10] Argonne National Laboratory, author of Otter, the first high-performance theorem prover. Many important papers, recipient of the Herbrand Award 2000.
- Hubert Comon ^[11], CNRS and now ENS Cachan. Many important papers.
- Robert Constable, Cornell University. Important contributions to type theory, NuPRL.
- Martin Davis ^[12], author of the "Handbook of Artificial Reasoning", co-inventor of the DPLL algorithm, recipient of the Herbrand Award 2005.
- Branden Fitelson ^[13] University of California at Berkeley. Work in automated discovery of shortest axiomatic bases for logic systems.
- Harald Ganzinger, co-developer of the superposition calculus, head of the MPI Saarbrücken, recipient of the Herbrand Award 2004 (posthumous).

- Michael Genesereth ^[14], Stanford University professor of Computer Science.
- Keith Goolsbey chief developer of the Cyc inference engine.
- Michael J. C. Gordon led the development of the HOL theorem prover.
- Gerard Huet ^[15] Term rewriting, HOL logics, Herbrand Award 1998
- Robert Kowalski developed the connection graph theorem-prover and SLD resolution, the inference engine that executes logic programs.
- Donald W. Loveland ^[16] Duke University. Author, co-developer of the DPLL-procedure, developer of model elimination, recipient of the Herbrand Award 2001.
- Norman Megill, developer of Metamath, and maintainer of its site at metamath.org ^[17], an online database of automatically verified proofs.
- J Strother Moore, co-author of the Boyer-Moore theorem prover, co-recipient of the Herbrand Award 1999.
- Robert Nieuwenhuis University of Barcelona. Co-developer of the superposition calculus.
- Tobias Nipkow Technical University of Munich, contributions to (higher-order) rewriting, co-developer of the Isabelle, proof assistant
- Ross Overbeek Argonne National Laboratory. Founder of The Fellowship for Interpretation of Genomes ^[18]
- Lawrence C. Paulson University of Cambridge, work on higher-order logic system, co-developer of the Isabelle proof assistant
- David A. Plaisted University of North Carolina at Chapel Hill. Complexity results, contributions to rewriting and completion, instance-based theorem proving.
- John Rushby ^[19] Program Director - SRI International
- J. Alan Robinson Syracuse University. Developed original resolution and unification based first order theorem proving, co-editor of the "Handbook of Automated Reasoning", recipient of the Herbrand Award 1996
- Jürgen Schmidhuber Work on Gödel Machines: Self-Referential Universal Problem Solvers Making Provably Optimal Self-Improvements ^[20]
- Stephan Schulz, E theorem Prover.
- Natarajan Shankar SRI International, work on decision procedures, *little engines of proof*, co-developer of PVS.
- Mark Stickel SRI. Recipient of the Herbrand Award 2002.
- Geoff Sutcliffe University of Miami. Maintainer of the TPTP collection, an organizer of the CADE annual contest.
- Dolph Ulrich ^[21] Purdue, Work on automated discovery of shortest axiomatic bases for systems.
- Robert Veroff ^[22] University of New Mexico. Many important papers.
- Andrei Voronkov ^[23] Developer of Vampire and Co-Editor of the "Handbook of Automated Reasoning"
- Larry Wos ^[24] Argonne National Laboratory. (Otter) Many important papers. Very first Herbrand Award winner (1992)
- Wen-Tsun Wu Work in geometric theorem proving: Wu's method, Herbrand Award 1997

See also

- Symbolic computation
- Computer-aided proof
- Automated reasoning
- Formal verification
- Logic programming
- Proof checking
- Model checking
- Proof complexity
- Computer algebra system
- Program analysis (computer science)

References

- Chin-Liang Chang; Richard Char-Tung Lee (1973). *Symbolic Logic and Mechanical Theorem Proving*. Academic Press1.
- Loveland, Donald W. (1978). *Automated Theorem Proving: A Logical Basis. Fundamental Studies in Computer Science Volume 6*. North-Holland Publishing.
- Gallier, Jean H. (1986). *Logic for Computer Science: Foundations of Automatic Theorem Proving*^[1]. Harper & Row Publishers (Available for free download).
- Duffy, David A. (1991). *Principles of Automated Theorem Proving*. John Wiley & Sons.
- Wos, Larry; Overbeek, Ross; Lusk, Ewing; Boyle, Jim (1992). *Automated Reasoning: Introduction and Applications* (2nd ed.). McGraw-Hill.
- Alan Robinson and Andrei Voronkov (eds.), ed (2001). *Handbook of Automated Reasoning Volume I & II*. Elsevier and MIT Press.
- Fitting, Melvin (1996). *First-Order Logic and Automated Theorem Proving*^[25] (2nd ed.). Springer.

References

- [1] W.W. McCune (1997). "Solution of the Robbins Problem" (<http://www.springerlink.com/content/h77246751668616h/>). *Journal of Automated Reasoning* **19** (3). .
- [2] Gina Kolata (December 10, 1996). "Computer Math Proof Shows Reasoning Power" (<http://www.nytimes.com/library/cyber/week/1210math.html>). The New York Times. . Retrieved 2008-10-11.
- [3] <http://www.cs.miami.edu/~tptp/>
- [4] <http://www.manchester.ac.uk/>
- [5] <http://www.cs.man.ac.uk/~voronkov/>
- [6] <http://www.freewebs.com/riazanov/>
- [7] Artosi, Alberto, Cattabriga, Paola and Governatori, Guido (1994-01-01). KED: A Deontic Theorem Prover. In: Biagioli, Carlo, Sartor, Giovanni and Tiscornia, Daniela Workshop on Legal Application of Logic Programming, Santa Margherita Ligure, Italy, (60-76).
- [8] <http://nevidal.org/sad.en.html>
- [9] <http://www.leoprover.org>
- [10] <http://www-unix.mcs.anl.gov/~mccune/>
- [11] <http://www.lsv.ens-cachan.fr/~comon/>
- [12] <http://www.cs.nyu.edu/cs/faculty/davism/>
- [13] <http://www.fitelson.org/>
- [14] <http://logic.stanford.edu/people/genesereth/>
- [15] <http://pauillac.inria.fr/~huet/>
- [16] <http://www.cs.duke.edu/~dwl/>
- [17] <http://www.metamath.org>
- [18] <http://theseed.uchicago.edu/FIG/Html/FIG.html>
- [19] <http://www.csl.sri.com/users/rushby/>
- [20] <http://www.idsia.ch/~juergen/goedelmachine.html>
- [21] <http://web.ics.purdue.edu/~dulrich/Home-page.htm>
- [22] <http://www.cs.unm.edu/~veroff/>
- [23] <http://www.voronkov.com/>
- [24] <http://www-unix.mcs.anl.gov/~wos/>
- [25] <http://comet.lehman.cuny.edu/fitting/>

Prover9

Prover9 is an automated theorem prover for First-order and equational logic developed by William McCune. Prover9 is the successor of the Otter theorem prover.

Prover9 is intentionally paired with Mace4, which searches for finite models and counterexamples. Both can be run simultaneously from the same input, with Prover9 attempting to find a proof, while Mace4 attempts to find a (disproving) counter-example. Prover9, Mace4, and many other tools are built on an underlying library named LADR to simplify implementation. Resulting proofs can be double-checked by Ivy, a proof-checking tool that has been separately verified using ACL2.

In July 2006 the LADR/Prover9/Mace4 input language made a major change (which also differentiates it from Otter). The key distinction between "clauses" and "formulas" completely disappeared; "formulas" can now have free variables; and "clauses" are now a subset of "formulas". Prover9/Mace4 also supports a "goal" type of formula, which is automatically negated for proof. Prover9 attempts to automatically generate a proof by default; in contrast, Otter's automatic mode must be explicitly set.

Prover9 is under active development, with new releases every month or every other month. Prover9 is free software/open source software; it is released under GPL version 2 or later.

Examples

Socrates

The traditional "all men are mortal", "Socrates is a man", prove "Socrates is mortal" can be expressed this way in Prover9:

```
formulas(assumptions).
    man(x) -> mortal(x).    % open formula with free variable x
    man(socrates).
end_of_list.
```

```
formulas(goals).
    mortal(socrates).
end_of_list.
```

This will be automatically converted into clausal form (which Prover9 also accepts):

```
formulas(sos).
    -man(x) | mortal(x).
    man(socrates).
    -mortal(socrates).
end_of_list.
```

Square Root of 2 is irrational

A proof that the square root of 2 is irrational can be expressed this way:

```
formulas(assumptions).
    1*x = x.                % identity
    x*y = y*x.              % commutativity
    x*(y*z) = (x*y)*z.      % associativity
    ( x*y = x*z ) -> y = z. % cancellation (0 is not allowed, so x!=0).
```

```

%
% Now let's define divides(x,y): x divides y.
% Example: divides(2,6) is true b/c 2*3=6.
%
divides(x,y) <-> (exists z x*z = y).
divides(2,x*x) -> divides(2,x).      % If 2 divides x*x, it divides x.
a*a = 2*(b*b).                      % a/b = sqrt(2), so a^2 = 2 * b^2.
(x != 1) -> -(divides(x,a) &
               divides(x,b)).        % a/b is in lowest terms
2 != 1.                             % Original author almost forgot this.
end_of_list.

```

External links

- Prover9 home page ^[1]
- Prover9 - Mace4 - LADR forums ^[2]
- Formal methods (square root of 2 example) ^[3]

References

- [1] <http://www.cs.unm.edu/~mccune/prover9/>
 [2] <http://forums.prover9.org/>
 [3] http://dwheeler.com/formal_methods/

Method of analytic tableaux

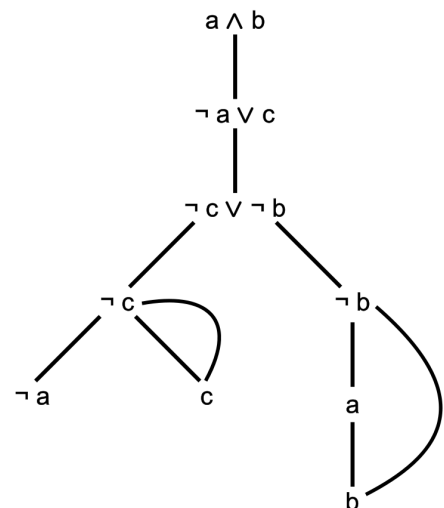
In proof theory, the **semantic tableau** (or **truth tree**) is a decision procedure for sentential and related logics, and a proof procedure for formulas of first-order logic. The tableau method can also determine the satisfiability of finite sets of formulas of various logics. It is the most popular proof procedure for modal logics (Girle 2000). The method of semantic tableaux was invented by the Dutch logician Evert Willem Beth.

An **analytic tableau** has for each node a subformula of the formula at the origin. In other words, it is a tableau satisfying the subformula property.

Introduction

For refutation tableaux, the objective is to show that the negation of a formula cannot be satisfied. There are rules for handling each of the usual connectives, starting with the main connective. In many cases, applying these rules causes the subtableau to divide into two. Quantifiers are instantiated. If any branch of a tableau leads to an evident contradiction, the branch *closes*. If all branches close, the proof is complete and the original formula is a logical truth.

Although the fundamental idea behind the **analytic tableau method** is derived from the cut-elimination theorem of structural proof theory, the origins of tableau calculi lie in the meaning (or semantics) of the logical connectives, as



A graphical representation of a partially built propositional tableau

the connection with proof theory was made only in recent decades.

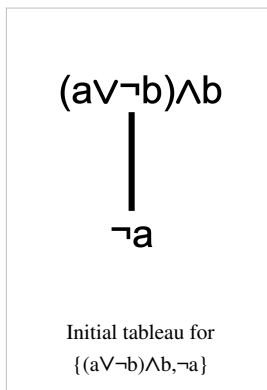
More specifically, a tableau calculus consists of a finite collection of rules with each rule specifying how to break down one logical connective into its constituent parts. The rules typically are expressed in terms of finite sets of formulae, although there are logics for which we must use more complicated data structures, such as multisets, lists, or even trees of formulas. Henceforth, "set" denotes any of {set, multiset, list, tree}.

If there is such a rule for every logical connective then the procedure will eventually produce a set which consists only of atomic formulae and their negations, which cannot be broken down any further. Such a set is easily recognizable as satisfiable or unsatisfiable with respect to the semantics of the logic in question. To keep track of this process, the nodes of a tableau itself are set out in the form of a tree and the branches of this tree are created and assessed in a systematic way. Such a systematic method for searching this tree gives rise to an algorithm for performing deduction and automated reasoning. Note that this larger tree is present regardless of whether the nodes contain sets, multisets, lists or trees.

Propositional logic

This section presents the tableau calculus for classical propositional logic. Tableau checks whether a given set of formulae is satisfiable or not. It can be used to check validity or entailment: a formula is valid if its negation is unsatisfiable and formulae A_1, \dots, A_n imply B if $\{A_1, \dots, A_n, \neg B\}$ is unsatisfiable.

The main principle of propositional tableau is to attempt to "break" complex formulae into smaller ones until complementary pairs of literals are produced or no further expansion is possible.



The method works on a tree whose nodes are labeled with formulae. At each step, this tree is modified; in the propositional case, the only allowed changes are additions of nodes as descendant of a leaf. The procedure starts by generating the tree made of a chain of all formulae in the set to prove unsatisfiability. A variant to this starting step is to begin with a single-node tree whose root is labeled by \top ; in this second case, the procedure can always copy a formula in the set below a leaf. As a running example, the tableau for the set $\{(a \vee \neg b) \wedge b, \neg a\}$ is shown.

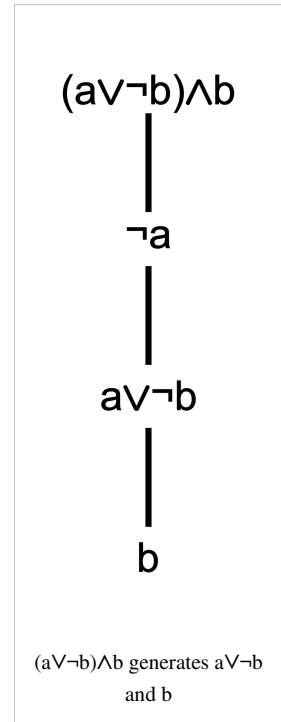
The principle of tableau is that formulae in nodes of the same branch are considered in conjunction while the different branches are considered to be disjunct. As a result, a tableau is a tree-like representation of a formula that is a disjunction of conjunctions. This formula is equivalent to the set to prove unsatisfiability. The procedure modifies the tableau in such a way that the formula represented by the resulting tableau is equivalent to the original one. One of these conjunctions may contain a pair of complementary literals, which proves that that conjunction is unsatisfiable. If all conjunctions are proved unsatisfiable, the original set of formulae is unsatisfiable.

And

Whenever a branch of a tableau contains a formula $A \wedge B$ that is the conjunction of two formulae, these two formulae are both consequences of that formula. This fact can be formalized by the following rule for expansion of a tableau:

(\wedge) If a branch of the tableau contains a conjunctive formula $A \wedge B$,
add to its leaf the chain of two nodes containing the formulae A and B

This rule is generally written as follows:



$$(\wedge) \frac{A \wedge B}{\begin{array}{c} A \\ B \end{array}}$$

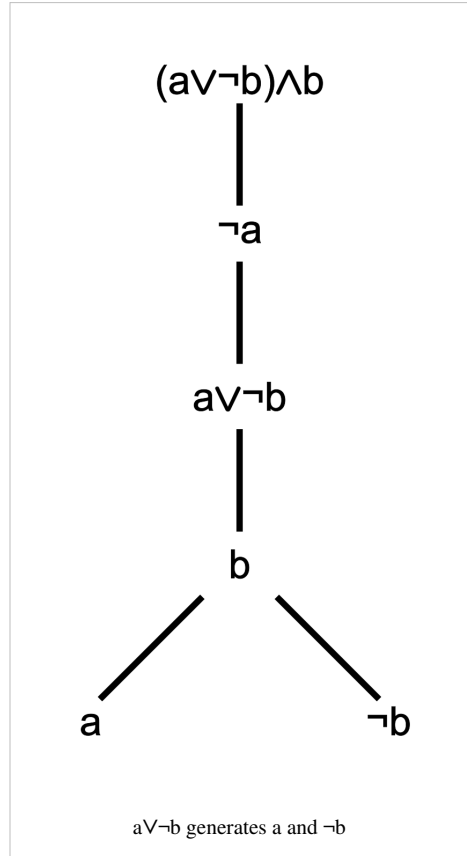
A variant of this rule allows a node to contain a set of formulae rather than a single one. In this case, the formulae in this set are considered in conjunction, so one can add $\{A, B\}$ at the end of a branch containing $A \wedge B$. More precisely, if a node on a branch is labeled $X \cup \{A \wedge B\}$, one can add to the branch the new leaf $X \cup \{A, B\}$.

Or

If a branch of a tableau contains a formula that is a disjunction of two formulae, such as $A \vee B$, the following rule can be applied:

(\vee) If a node on a branch contains a disjunctive formula $A \vee B$, then create two sibling children to the leaf of the branch, containing the formulae A and B , respectively.

This rule splits a branch into two ones, differing only for the final node. Since branches are considered in disjunction to each other, the two resulting branches are equivalent to the original one, as the disjunction of their non-common nodes is precisely $A \vee B$. The rule for disjunction is generally formally written using the symbol $|$ for separating the formulae of the two distinct nodes to be created:



$$(\vee) \frac{A \vee B}{A | B}$$

If nodes are assumed to contain sets of formulae, this rule is replaced by: if a node is labeled $Y \cup \{A \vee B\}$, a leaf of the branch this node is in can be appended two sibling child nodes labeled $Y \cup \{A\}$ and $Y \cup \{B\}$, respectively.

Not

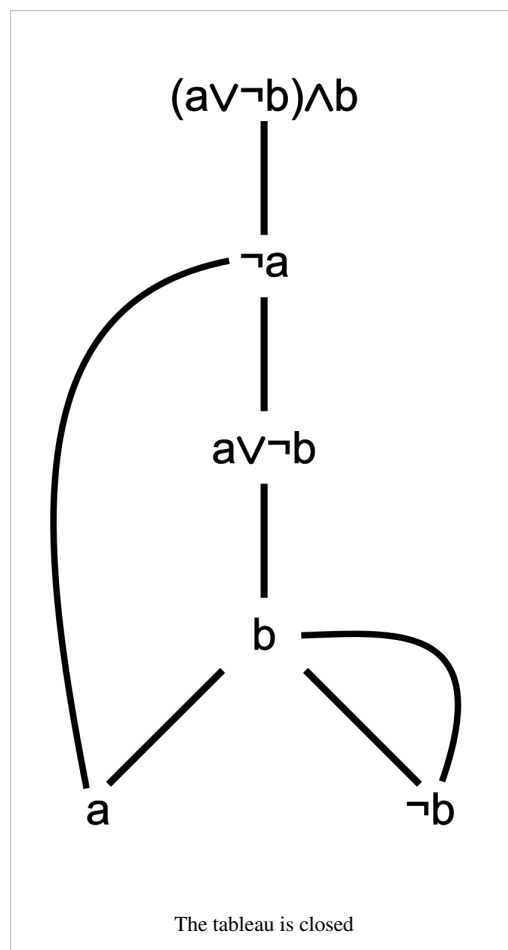
The aim of tableau is to generate progressively simpler formulae until pairs of opposite literals are produced or no other rule can be applied. Negation can be treated by initially making formulae in negation normal form, so that negation only occurs in front of literals. Alternatively, one can use De Morgan's laws during the expansion of the tableau, so that for example $\neg(A \wedge B)$ is treated as $\neg A \vee \neg B$. Rules that introduce or remove a pair of negations (such as in $\neg\neg A$) are also used in this case (otherwise, there would be no way of expanding a formula like $\neg\neg(A \wedge B)$):

$$(\neg 1) \frac{A}{\neg\neg A}$$

$$(\neg 2) \frac{\neg\neg A}{A}$$

Closure

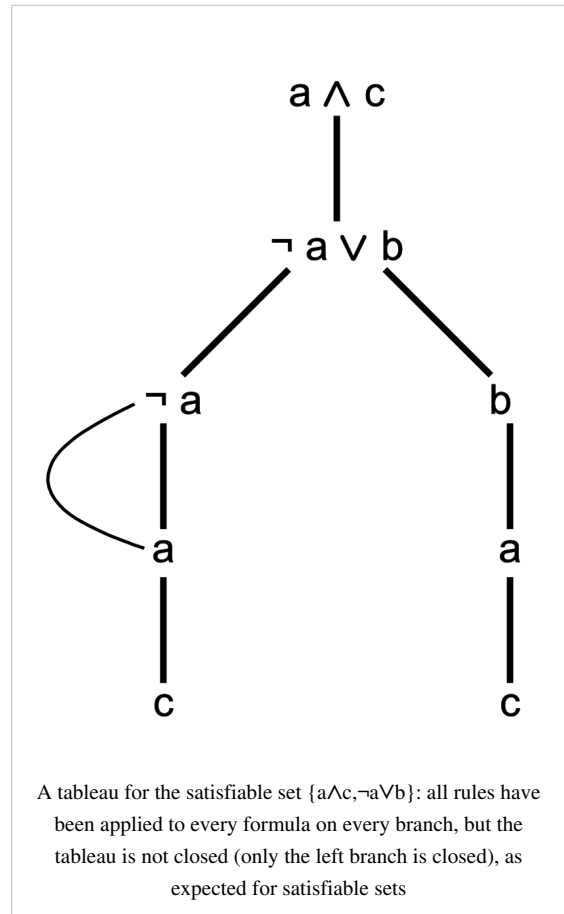
Every tableau can be considered as a graphical representation of a formula, which is equivalent to the set the tableau is built from. This formula is as follows: each branch of the tableau represents the conjunction of its formulae; the tableau represents the disjunction of its branches. The expansion rules transform a tableau into one having an equivalent represented formula. Since the tableau is initialized as a single branch containing the formulae of the input set, all subsequent tableaux obtained from it represent formulae which are equivalent to that set (in the variant where the initial tableau is the single node labeled true, the formulae represented by tableaux are consequences of the original set.)



The method of tableaux works by starting with the initial set of formulae and then adding to the tableau simpler and simpler formulae, while still preserving equivalent to the original set, until contradiction is shown in the simple form of opposite literals. Since the formula represented by a tableau is the disjunction of the formulae represented by its branches, contradiction is obtained when every branch contains a pair of opposite literals.

Once a branch contains a literal and its negation, its corresponding formula is unsatisfiable. As a result, this branch can be now "closed", as there is no need to further expand it. If all branches of a tableau are closed, the formula represented by the tableau is unsatisfiable; therefore, the original set is unsatisfiable as well. Obtaining a tableau where all branches are closed is a way for proving the unsatisfiability of the original set. In the propositional case, one can also prove that satisfiability is proved by the impossibility of finding a closed tableau, provided that every expansion rule has been applied everywhere it could be applied. In particular, if a tableau contains some open (non-closed) branches and every formula that is not a literal has been used by a rule to generate a new node on every branch the formula is in, the set is satisfiable.

This rule takes into account that a formula may occur in more than one branch (this is the case if there is at least a branching point "below" the node). In this case, the rule for expanding the formula has to be applied so that its conclusion(s) are appended to all of these branches that are still open, before one can conclude that the tableau cannot be further expanded and that the formula is therefore satisfiable.



Set-labeled tableau

A variant of tableau is to label nodes with sets of formulae rather than single formulae. In this case, the initial tableau is a single node labeled with the set to be proved satisfiable. The formulae in a set are therefore considered to be in conjunction.

The rules of expansion of the tableau can now work on the leaves of the tableau, ignoring all internal nodes. For conjunction, the rule is based on the equivalence of a set containing a conjunction $A \wedge B$ with the set containing both A and B in place of it. In particular, if a leaf is labeled with $X \cup \{A \wedge B\}$, a node can be appended to it with label $X \cup \{A, B\}$:

$$(\wedge) \frac{X \cup \{A \wedge B\}}{X \cup \{A, B\}}$$

For disjunction, a set $X \cup \{A \vee B\}$ is equivalent to the disjunction of the two sets $X \cup \{A\}$ and $X \cup \{B\}$. As a result, if the first set labels a leaf, two children can be appended to it, labeled with the latter two formulae.

$$(\vee) \frac{X \cup \{A \vee B\}}{X \cup \{A\} \mid X \cup \{B\}}$$

Finally, if a set contains both a literal and its negation, this branch can be closed:

$$(id) \frac{X \cup \{p, \neg p\}}{closed}$$

A tableau for a given finite set X is a finite (upside down) tree with root X in which all child nodes are obtained by applying the tableau rules to their parents. A branch in such a tableau is closed if its leaf node contains "closed". A tableau is closed if all its branches are closed. A tableau is open if at least one branch is not closed.

Here are two closed tableaux for the set $X = \{r0 \ \& \ \sim r0, p0 \ \& \ ((\sim p0 \vee q0) \ \& \ \sim q0)\}$ with each rule application marked at the right hand side (& and \sim stand for \wedge and \neg , respectively)

| | |
|--|--|
| <pre> {r0 & ~r0, p0 & ((~p0 v q0) & ~q0)} ----- (&) {r0 , ~r0, p0 & ((~p0 v q0) & ~q0)} ----- (id) closed </pre> | <pre> {r0 & ~r0, p0 & ((~p0 v q0) & ~q0)} ----- (&) {r0 & ~r0, p0, ((~p0 v q0) & ~q0)} ----- (&) {r0 & ~r0, p0, (~p0 v q0), ~q0} ----- (v) {r0 & ~r0, p0, ~p0, ~q0} {r0 & ~r0, p0, q0, ~q0} ----- (id) ----- (id) closed closed </pre> |
|--|--|

The left hand tableau closes after only one rule application while the right hand one misses the mark and takes a lot longer to close. Clearly, we would prefer to always find the shortest closed tableaux but it can be shown that one single algorithm that finds the shortest closed tableaux for all input sets of formulae cannot exist.

The three rules (\wedge) , (\vee) and (id) given above are then enough to decide if a given set X' of formulae in negated normal form are jointly satisfiable:

Just apply all possible rules in all possible orders until we find a closed tableau for X' or until we exhaust all possibilities and conclude that every tableau for X' is open.

In the first case, X' is jointly unsatisfiable and in the second the case the leaf node of the open branch gives an assignment to the atomic formulae and negated atomic formulae which makes X' jointly satisfiable. Classical logic actually has the rather nice property that we need to investigate only (any) one tableau completely: if it closes then X' is unsatisfiable and if it is open then X' is satisfiable. But this property is not generally enjoyed by other logics. These rules suffice for all of classical logic by taking an initial set of formulae X and replacing each member C by its logically equivalent negated normal form C' giving a set of formulae X' . We know that X is satisfiable if and only if X' is satisfiable, so it suffices to search for a closed tableau for X' using the procedure outlined above.

By setting $X = \{\neg A\}$ we can test whether the formula A is a tautology of classical logic:

If the tableau for $\{\neg A\}$ closes then $\neg A$ is unsatisfiable and so A is a tautology since no assignment of truth values will ever make A false. Otherwise any open leaf of any open branch of any open tableau for $\{\neg A\}$ gives an assignment that falsifies A .

Conditional

Classical propositional logic usually has a connective to denote material implication. If we write this connective as \Rightarrow , then the formula $A \Rightarrow B$ stands for "if A then B ". It is possible to give a tableau rule for breaking down $A \Rightarrow B$ into its constituent formulae. Similarly, we can give one rule each for breaking down each of $\neg(A \wedge B)$, $\neg(A \vee B)$, $\neg(\neg A)$, and $\neg(A \Rightarrow B)$. Together these rules would give a terminating procedure for deciding whether a given set of formulae is simultaneously satisfiable in classical logic since each rule breaks down one formula into its constituents but no rule builds larger formulae out of smaller constituents. Thus we must eventually reach a node that contains only atoms and negations of atoms. If this last node matches (id) then we can close the branch, otherwise it remains open.

But note that the following equivalences hold in classical logic where $(...) = (...)$ means that the left hand side formula is logically equivalent to the right hand side formula:

$$\begin{aligned}
\neg(A \wedge B) &= \neg A \vee \neg B \\
\neg(A \vee B) &= \neg A \wedge \neg B \\
\neg(\neg A) &= A \\
\neg(A \Rightarrow B) &= A \wedge \neg B \\
A \Rightarrow B &= \neg A \vee B \\
A \Leftrightarrow B &= (A \wedge B) \vee (\neg A \wedge \neg B) \\
\neg(A \Leftrightarrow B) &= (A \wedge \neg B) \vee (\neg A \wedge B)
\end{aligned}$$

If we start with an arbitrary formula C of classical logic, and apply these equivalences repeatedly to replace the left hand sides with the right hand sides in C , then we will obtain a formula C' which is logically equivalent to C but which has the property that C' contains no implications, and \neg appears in front of atomic formulae only. Such a formula is said to be in negation normal form and it is possible to prove formally that every formula C of classical logic has a logically equivalent formula C' in negation normal form. That is, C is satisfiable if and only if C' is satisfiable.

First-order logic tableau

Tableaux are extended to first order predicate logic by two rules for dealing with universal and existential quantifiers, respectively. Two different sets of rules can be used; both employ a form of Skolemization for handling existential quantifiers, but differ on the handling of universal quantifiers.

The set of formulae to check for validity is here supposed to contain no free variables; this is not a limitation as free variables are implicitly universally quantified, so universal quantifiers over these variables can be added, resulting in a formula with no free variables.

First-order tableau without unification

A first-order formula $\forall x.\gamma(x)$ implies all formulae $\gamma(t)$ where t is a ground term. The following inference rule is therefore correct:

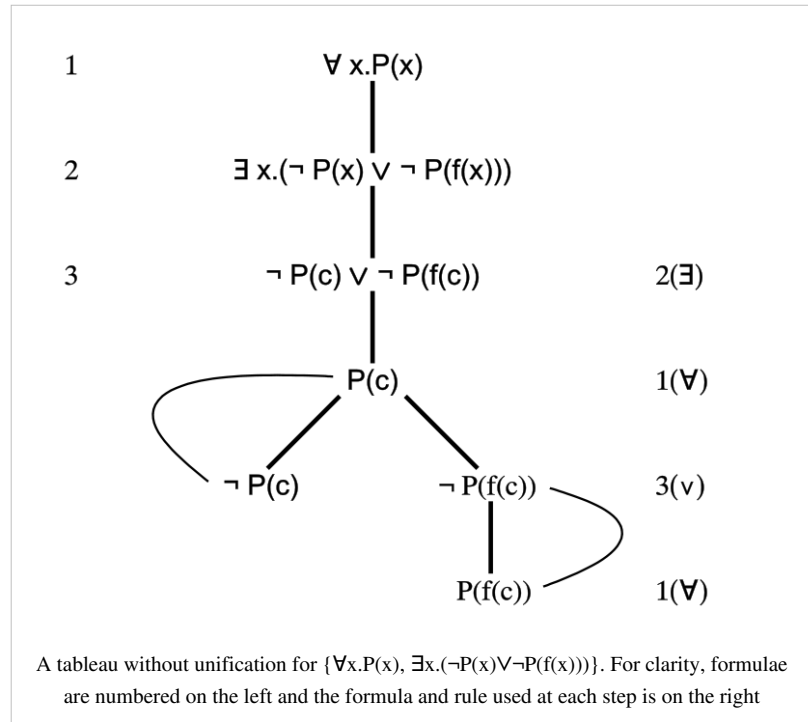
$$(\forall) \frac{\forall x.\gamma(x)}{\gamma(t)} \text{ where } t \text{ is an arbitrary ground term}$$

Contrarily to the rules for the propositional connectives, multiple applications of this rule to the same formula may be necessary. As an example, the set $\{\neg P(a) \vee \neg P(b), \forall x.P(x)\}$ can only be proved unsatisfiable if both $P(a)$ and $P(b)$ are generated from $\forall x.P(x)$.

Existential quantifiers are dealt with Skolemization. In particular, a formula with a leading existential quantifier like $\exists x.\delta(x)$ generates its Skolemization $\delta(c)$, where c is a new constant symbol.

$$(\exists) \frac{\exists x.\delta(x)}{\delta(c)} \text{ where } c \text{ is a new constant symbol}$$

The Skolem term c is a constant (a function of arity 0) because the quantification over x does not occur within the scope of any universal quantifier. If the original formula contained some universal quantifiers such that the quantification over x was within their scope, these quantifiers have evidently been removed by the application of the rule for universal quantifiers.



The rule for existential quantifiers introduces new constant symbols. These symbols can be used by the rule for universal quantifiers, so that $\forall y.\gamma(y)$ can generate $\gamma(c)$ even if c was not in the original formula but is a Skolem constant created by the rule for existential quantifiers.

The above two rules for universal and existential quantifiers are correct, and so are the propositional rules: if a set of formulae generates a closed tableau, this set is unsatisfiable. Completeness can also be proved: if a set of formulae is unsatisfiable, there exists a closed tableau built from it by these rules. However, actually finding such a closed tableau requires a suitable policy of application of rules. Otherwise, an unsatisfiable set can generate an infinite-growing tableau. As an example, the set $\{\neg P(f(c)), \forall x.P(x)\}$ is unsatisfiable, but a closed tableau is never obtained if one unwisely keeps applying the rule for universal quantifiers to $\forall x.P(x)$, generating for example $P(c), P(f(c)), P(f(f(c))), \dots$. A closed tableau can always be found by ruling out this and similar "unfair" policies of application of tableau rules.

The rule for universal quantifiers (\forall) is the only non-deterministic rule, as it does not specify which term to instantiate with. Moreover, while the other rules need to be applied only once for each formula and each path the formula is in, this one may require multiple applications. Application of this rule can however be restricted by delaying the application of the rule until no other rule is applicable and by restricting the application of the rule to ground terms that already appear in the path of the tableau. The variant of tableaux with unification shown below aims at solving the problem of non-determinism.

First-order tableau with unification

The main problem of tableau without unification is how to choose a ground term t for the universal quantifier rule. Indeed, every possible ground term can be used, but clearly most of them might be useless for closing the tableau.

A solution to this problem is to "delay" the choice of the term to the time when the consequent of the rule allows closing at least a branch of the tableau. This can be done by using a variable instead of a term, so that $\forall x.\gamma(x)$ generates $\gamma(x')$, and then allowing substitutions to later replace x' with a term. The rule for universal quantifiers becomes:

$$(\forall) \frac{\forall x. \gamma(x)}{\gamma(x')} \text{ where } x' \text{ is a variable not occurring everywhere else in the tableau}$$

While the initial set of formulae is supposed not to contain free variables, a formula of the tableau contain the free variables generated by this rule. These free variables are implicitly considered universally quantified.

This rule employs a variable instead of a ground term. The gain of this change is that these variables can be then given a value when a branch of the tableau can be closed, solving the problem of generating terms that might be useless.

(σ) if σ is the most general unifier of two literals A and B , where A and the negation of B occur in the same branch of the tableau, σ can be applied at the same time to all formulae of the tableau

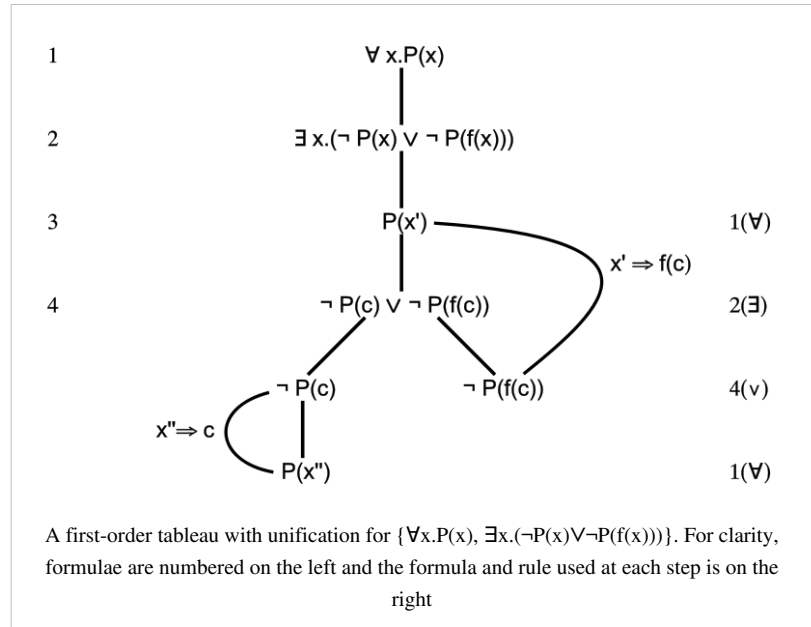
As an example, $\{\neg P(a), \forall x. P(x)\}$ can be proved unsatisfiable by first generating $P(x_1)$; the negation of this literal is unifiable with $\neg P(a)$, the most general unifier being the substitution that replaces x_1 with a ; applying this substitution results in replacing $P(x_1)$ with $P(a)$, which closes the tableau.

This rule closes at least a branch of the tableau -the one containing the considered pair of literals. However, the substitution has to be applied to the whole tableau, not only on these two literals. This is expressed by saying that the free variables of the tableau are *rigid*: if an occurrence of a variable is replaced by something else, all other occurrences of the same variable must be replaced in the same way. Formally, the free variables are (implicitly) universally quantified and all formulae of the tableau are within the scope of these quantifiers.

Existential quantifiers are dealt with by Skolemization. Contrary to the tableau without unification, Skolem terms may not be simple constant. Indeed, formulae in a tableau with unification may contain free variables, which are implicitly considered universally quantified. As a result, a formula like $\exists x. \delta(x)$ may be within the scope of universal quantifiers; if this is the case, the Skolem term is not a simple constant but a term made of a new function symbol and the free variables of the formula.

$$(\exists) \frac{\exists x. \delta(x)}{\delta(f(x_1, \dots, x_n))} \text{ where } f \text{ is a new function symbol and } x_1, \dots, x_n \text{ the free variables of } \delta$$

This rule incorporates a simplification over a rule where x_1, \dots, x_n are the free variables of the branch, not of δ alone. This rule can be further simplified by the reuse of a function symbol if it has already been used in a formula that is identical to δ up to variable renaming.



The formula represented by a tableau is obtained in a way that is similar to the propositional case, with the additional assumption that free variables are considered universally quantified. As for the propositional case, formulae in each branch are conjoined and the resulting formulae are disjoined. In addition, all free variables of the resulting formula are universally quantified. All these quantifiers have the whole formula in their scope. In other words, if F is the

formula obtained by disjoining the conjunction of the formulae in each branch, and x_1, \dots, x_n are the free variables in it, then $\forall x_1$ the formula represented by the tableau. The following considerations apply:

- The assumption that free variables are universally quantified is what makes the application of a most general unifier a sound rule: since $\gamma(x')$ means that γ is true for every possible value of x' , then $\gamma(t)$ is true for the term t that the most general unifier replaces x with.
- Free variables in a tableau are rigid: all occurrences of the same variable have to be replaced all with the same term. Every variable can be considered a symbol representing a term that is yet to be decided. This is a consequence of free variables being assumed universally quantified over the whole formula represented by the tableau: if the same variable occurs free in two different nodes, both occurrences are in the scope of the same quantifier. As an example, if the formulae in two nodes are $A(x)$ and $B(x)$, where x is free in both, the formula represented by the tableau is something in the form $\forall x.(\dots A(x) \dots B(x) \dots)$. This formula implies that $(\dots A(x) \dots B(x) \dots)$ is true for any value of x , but does not in general imply $(\dots A(t) \dots A(t') \dots)$ for two different terms t and t' , as these two terms may in general take different values. This means that x cannot be replaced by two different terms in $A(x)$ and $B(x)$.
- Free variables in a formula to check for validity are also considered universally quantified. However, these variables cannot be left free when building a tableau, because tableau rules work on the converse of the formula but still treat free variables as universally quantified. For example, $P(x) \rightarrow P(c)$ is not valid (it is not true in the model where $D = \{1, 2\}$, $P(1) = \perp$, $P(2) = \top$, $c = 1$, and the interpretation where $x = 2$). Consequently, $\{P(x), \neg P(c)\}$ is satisfiable (it is satisfied by the same model and interpretation). However, a closed tableau could be generated with $P(x)$ and $\neg P(c)$, and substituting x with c would generate a closure.

A correct procedure is to first make universal quantifiers explicit, thus generating $\forall x.(P(x) \rightarrow P(c))$. The following two variants are also correct.

- Applying to the whole tableau a substitution to the free variables of the tableau is a correct rule, provided that this substitution is free for the formula representing the tableau. In other words, applying such a substitution leads to a tableau whose formula is still a consequence of the input set. Using most general unifiers automatically ensures that the condition of freeness for the tableau is met.
- While in general every variable has to be replaced with the same term in the whole tableau, there are some special cases in which this is not necessary.

Tableaux with unification can be proved complete: if a set of formulae is unsatisfiable, it has a tableau-with-unification proof. However, actually finding such a proof may be a difficult problem. Contrarily to the case without unification, applying a substitution can modify the existing part of a tableau; while applying a substitution closes at least a branch, it may make other branches impossible to close (even if the set is unsatisfiable).

A solution to this problem is that *delayed instantiation*: no substitution is applied until one that closes all branches at the same time is found. With this variant, a proof for an unsatisfiable set can always be found by a suitable policy of application of the other rules. This method however requires the whole tableau to be kept in memory: the general method closes branches which can be then discarded, while this variant does not close any branch until the end.

The problem that some tableaux that can be generated are impossible to close even if the set is unsatisfiable is common to other sets of tableau expansion rules: even if some specific sequences of application of these rules allow constructing a closed tableau (if the set is unsatisfiable), some other sequences lead to tableau that cannot be closed. General solutions for these cases are outlined in the "Searching for a tableau" section.

Tableau calculi and their properties

A tableau calculus is a set of rules that allows building and modification of a tableau. Propositional tableau rules, tableau rules without unification, and tableau rules with unification, are all tableau calculi. Some important properties a tableau calculus may or may not possess are completeness, destructiveness, and proof confluence.

A tableau calculi is said complete if it allows building a tableau proof for every given unsatisfiable set of formulae. The tableau calculi mentioned above can be proved complete.

A remarkable difference between tableau with unification and the other two calculi is that the latter two calculi only modify a tableau by adding new nodes to it, while the former one allows substitutions to modify the existing part of the tableau. More generally, tableau calculi are classed as *destructive* or *non-destructive* depending on whether they only add new nodes to tableau or not. Tableau with unification is therefore destructive, while propositional tableau and tableau without unification are non-destructive.

Proof confluence is the property of a tableau calculus to obtain a proof for an arbitrary unsatisfiable set from an arbitrary tableau, assuming that this tableau has itself been obtained by applying the rules of the calculus. In other words, in a proof confluent tableau calculus, from an unsatisfiable set one can apply whatever set of rules and still obtain a tableau from which a closed one can be obtained by applying some other rules.

Proof procedures

A tableau calculus is simply a set of rules that tells how a tableau can be modified. A proof procedure is a method for actually finding a proof (if one exists). In other words, a tableau calculus is a set of rules, while a proof procedure is a policy of application of these rules. Even if a calculus is complete, not every possible choice of application of rules leads to a proof of an unsatisfiable set. For example $\{P(f(x)), R(c), \neg P(f(c)) \vee \neg R(c), \forall x.Q(x)\}$ is unsatisfiable, but both tableaux with unification and tableaux without unification allow the rule for the universal quantifiers to be applied repeatedly to the last formula, while simply applying the rule for disjunction to the third one would directly lead to closure.

For proof procedures, a definition of completeness has been given: a proof procedure is strongly complete if it allows finding a closed tableau for any given unsatisfiable set of formulae. Proof confluence of the underlying calculus is relevant to completeness: proof confluence is the guarantee that a closed tableau can be always generated from an arbitrary partially constructed tableau (if the set is unsatisfiable). Without proof confluence, the application of a 'wrong' rule may result in the impossibility of making the tableau complete by applying other rules.

Propositional tableaux and tableaux without unification have strongly complete proof procedures. In particular, a complete proof procedure is that of applying the rules in a *fair* way. This is because the only way such calculi cannot generate a closed tableau from an unsatisfiable set is by not applying some applicable rules.

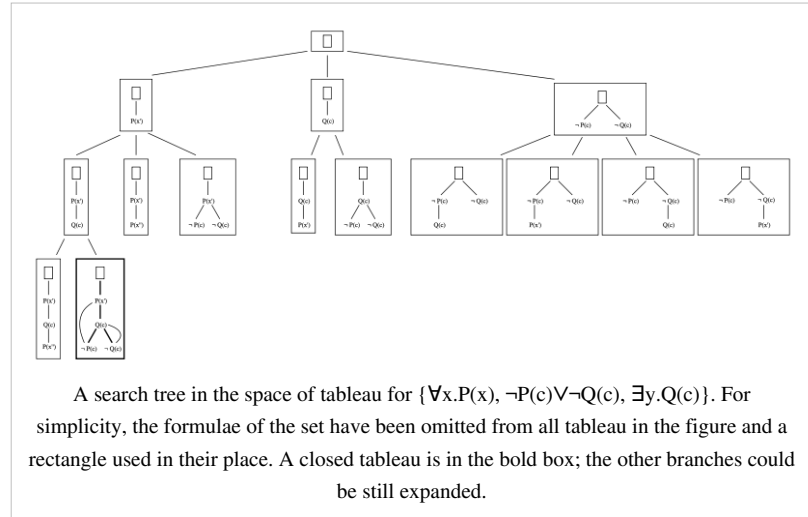
For propositional tableaux, fairness amounts to expanding every formula in every branch. More precisely, for every formula and every branch the formula is in, the rule having the formula as a precondition has been used to expand the branch. A fair proof procedure for propositional tableaux is strongly complete.

For first-order tableaux without unification, the condition of fairness is similar, with the exception that the rule for universal quantifier might require more than one application. Fairness amounts to expanding every universal quantifier infinitely often. In other words, a fair policy of application of rules cannot keep applying other rules without expanding every universal quantifier in every branch that is still open once in a while.

Searching for a closed tableau

If a tableau calculus is complete, every unsatisfiable set of formula has an associated closed tableau. While this tableau can always be obtained by applying some of the rules of the calculus, the problem of which rules to apply for a given formula still remains. As a result, completeness does not automatically implies the existence of a feasible policy of application of rules that can always lead to building a closed tableau for every given unsatisfiable set of formulae. While a fair proof procedure is complete for ground tableau and tableau without unification, this is not the case for tableau with unification.

A general solution for this problem is that of searching the space of tableau until a closed one is found (if any exists, that is, the set is unsatisfiable). In this approach, one starts with an empty tableau and then recursively tries to apply every possible applicable rule. This procedure visit a (implicit) tree whose nodes are labeled with tableau, and such that the tableau in a node is obtained from the tableau in its parent by applying one of the valid rules.



Since one such branch can be infinite, this tree has to be visited breadth-first rather than depth-first. This requires a large amount of space, as the breadth of the tree can grow exponentially. A method that may visit some nodes more than once but works in polynomial space is to visit in a depth-first manner with iterative deepening: one first visits the tree up to a certain depth, then increases the depth and perform the visit again. This particular procedure uses the depth (which is also the number of tableau rules that have been applied) for deciding when to stop at each step. Various other parameters (such as the size of the tableau labeling a node) have been used instead.

Reducing search

The size of the search tree depends on the number of (children) tableau that can be generated from a given (parent) one. Reducing the number of such tableau therefore reduces the required search.

A way for reducing this number is to disallow the generation of some tableau based on their internal structure. An example is the condition of regularity: if a branch contains a literal, using an expansion rule that generates the same literal is useless because the branch containing two copies of the literals would have the same set of formulae of the original one. This expansion can be disallowed because if a closed tableau exists, it can be found without it. This restriction is structural because it can be checked by looking at the structure of the tableau to expand only.

Different methods for reducing search disallow the generation of some tableau on the ground that a closed tableau can still be found by expanding the other ones. These restrictions are called global. As an example of a global restriction, one may employ a rule that specify which of the open branches is to be expanded. As a result, if a tableau has for example two non-closed branches, the rule tells which one is to be expanded, disallowing the expansion of the second one. This restriction reduces the search space because one possible choice is now forbidden; completeness is however not harmed, as the second branch will still be expanded if the first one is eventually closed. As an example, a tableau with root $\neg a \wedge \neg b$, child $a \vee b$, and two leaves a and b can be closed in two ways: applying (\wedge) first to a and then to b , or vice versa. There is clearly no need to follow both possibilities; one may consider only the case in which (\wedge) is first applied to a and disregard the case in which it is first applied to b . This is a global restriction because what allows neglecting this second expansion is the presence of the other tableau,

where expansion is applied to a first and b afterwards.

Clause tableaux

When applied to sets of clauses (rather than of arbitrary formulae), tableaux methods allow for a number of efficiency improvements. A first-order clause is a formula $\forall x_1, \dots, x_n L_1 \vee \dots \vee L_m$ that does not contain free variables and such that each L_i is a literal. The universal quantifiers are often omitted for clarity, so that for example $P(x, y) \vee Q(f(x))$ actually means $\forall x, y. P(x, y) \vee Q(f(x))$. Note that, if taken literally, these two formulae are not the same as for satisfiability: rather, the satisfiability $P(x, y) \vee Q(f(x))$ is the same as that of $\exists x, y. P(x, y) \vee Q(f(x))$. That free variables are universally quantified is not a consequence of the definition of first-order satisfiability; it is rather used as an implicit common assumption when dealing with clauses. The only expansion rules that are applicable to a clause are (\forall) and (\vee) ; these two rules can be replaced by their combination without losing completeness. In particular, the following rule corresponds to applying in sequence the rules (\forall) and (\vee) of the first-order calculus with unification.

$$(C) \frac{L_1 \vee \dots \vee L_n}{L'_1 \vee \dots \vee L'_n} \text{ where } L'_1 \vee \dots \vee L'_n \text{ is obtained by replacing every variable with a new one in } L_1 \vee \dots \vee L_n$$

When the set to be checked for satisfiability is only composed of clauses, this and the unification rules are sufficient to prove unsatisfiability. In other words, the tableau calculi composed of (C) and (σ) is complete.

Since the clause expansion rule only generates literals and never new clauses, the clauses to which it can be applied are only clauses of the input set. As a result, the clause expansion rule can be further restricted to the case where the clause is in the input set.

$$(C) \frac{L_1 \vee \dots \vee L_n}{L'_1 \vee \dots \vee L'_n} \text{ where } L'_1 \vee \dots \vee L'_n \text{ is obtained by replacing every variable with a}$$

new one in $L_1 \vee \dots \vee L_n$, which is a clause of the input set

Since this rule directly exploits the clauses in the input set there is no need to initialize the tableau to the chain of the input clauses. The initial tableau can therefore be initialized with the single node labeled *true*; this label is often omitted as implicit. As a result of this further simplification, every node of the tableau (apart of the root) is labeled with a literal.

A number of optimizations can be used for clause tableau. These optimizations are aimed at reducing the number of possible tableaux to be explored when searching for a closed tableau as described in the "Searching for a closed tableau" section above.

Connection tableau

Connection is a condition over tableau that forbids expanding a branch using clauses that are unrelated to the literals that are already in the branch. Connection can be defined in two ways:

strong connectedness

when expanding a branch, use an input clause only if it contains a literal that can be unified with the negation of the literal in the current leaf

weak connectedness

allow the use of clauses that contain a literal that unifies with the negation of a literal on the branch

Both conditions apply only to branches consisting not only of the root. The second definition allows for the use of a clause containing a literal that unifies with the negation of a literal in the branch, while the first only further constraint that literal to be in leaf of the current branch.

If clause expansion is restricted by connectedness (either strong or weak), its application produces a tableau in which substitution can be applied to one of the new leaves, closing its branch. In particular, this is the leaf containing the literal of the clause that unifies with the negation of a literal in the branch (or the negation of the literal in the parent, in case of strong connection).

Both conditions of connectedness lead to a complete first-order calculus: if a set of clauses is unsatisfiable, it has a closed connected (strongly or weakly) tableau. Such a closed tableau can be found by searching in the space of tableaux as explained in the "Searching for a closed tableau" section. During this search, connectedness eliminates some possible choices of expansion, thus reducing search. In other words, while the tableau in a node of the tree can be in general expanded in several different ways, connection may allow only few of them, thus reducing the number of resulting tableaux that need to be further expanded.

This can be seen on the following (propositional) example. The tableau made of a chain $true - a$ for the set of clauses $\{a, \neg a \vee b, \neg c \vee d, \neg b\}$ can be in general expanded using each of the four input clauses, but connection only allows the expansion that uses $\neg a \vee b$. This means that the tree of tableaux has four leaves in general but only one if connectedness is imposed. This means that connectedness leaves only one tableau to try and expand, instead of the four ones to consider in general. In spite of this reduction of choices, the completeness theorem implies that a closed tableau can be found if the set is unsatisfiable.

The connectedness conditions, when applied to the propositional (clausal) case, make the resulting calculus non-confluent. As an example, $\{a, b, \neg b\}$ is unsatisfiable, but applying (C) to a generates the chain $true - a$, which is not closed and to which no other expansion rule can be applied without violating either strong or weak connectedness. In the case of weak connectedness, confluence holds provided that the clause used for expanding the root is relevant to unsatisfiability, that is, it is contained in a minimally unsatisfiable subset of the set of clauses. Unfortunately, the problem of checking whether a clause meets this condition is itself a hard problem. In spite of non-confluence, a closed tableau can be found using search, as presented in the "Searching for a closed tableau" section above. While search is made necessary, connectedness reduces the possible choices of expansion, thus making search more efficient.

Regular tableaux

A tableau is regular if no literal occurs twice in the same branch. Enforcing this condition allows for a reduction of the possible choices of tableau expansion, as the clauses that would generate a non-regular tableau cannot be expanded.

These disallowed expansion steps are however useless. If B is a branch containing a literal L , and C is a clause whose expansion violates regularity, then C contains L . In order to close the tableau, one needs to expand and close, among others, the branch where $B - L$, where L occurs twice. However, the formulae in this branch are exactly the same as the formulae of B alone. As a result, the same expansion steps that close $B - L$ also close B . This means that expanding C was unnecessary; moreover, if C contained other literals, its expansion generated other leaves that needed to be closed. In the propositional case, the expansion needed to close these leaves are completely useless; in the first-order case, they may only affect the rest of the tableau because of some unifications; these can however be combined to the substitutions used to close the rest of the tableau.

Tableaux for modal logics

In a modal logic, a model comprises a set of *possible worlds*, each one associated to a truth evaluation; an *accessibility relation* tells when a world is *accessible* from another one. A modal formula may specify not only conditions over a possible world, but also on the ones that are accessible from it. As an example, $\Box A$ is true in a world if A is true in all worlds that are accessible from it.

As for propositional logic, tableaux for modal logics are based on recursively breaking formulae into its basic components. Expanding a modal formula may however require stating conditions over different worlds. As an example, if $\neg\Box A$ is true in a world then there exists a world accessible from it where A is false. However, one cannot simply add the following rule to the propositional ones.

$$\frac{\neg\Box A}{\neg A}$$

In propositional tableaux all formulae refer to the same truth evaluation, but the precondition of the rule above holds in a world while the consequence holds in another. Not taking into account this would generate wrong results. For example, formula $a \wedge \neg\Box a$ states that a is true in the current world and a is false in a world that is accessible from it. Simply applying (\wedge) and the expansion rule above would produce a and $\neg a$, but these two formulae should not in general generate a contradiction, as they hold in different worlds. Modal tableaux calculi do contain rules of the kind of the one above, but include mechanisms to avoid the incorrect interaction of formulae referring to different worlds.

Technically, tableaux for modal logics check the satisfiability of a set of formulae: they check whether there exists a model M and world w such that the formulae in the set are true in that model and world. In the example above, while a states the truth of a in w , the formula $\neg\Box a$ states the truth of $\neg a$ in some world w' that is accessible from w and which may in general be different from w . Tableaux calculi for modal logic take into account that formulae may refer to different worlds.

This fact has an important consequence: formulae that hold in a world may imply conditions over different successors of that world. Unsatisfiability may then be proved from the subset of formulae referring to the a single successor. This holds if a world may have more than one successor, which is true for most modal logic. If this is the case, a formula like $\neg\Box A \wedge \neg\Box B$ is true if a successor where $\neg A$ holds exists and a successor where $\neg B$ holds exists. In the other way around, if one can show unsatisfiability of $\neg A$ in an arbitrary successor, the formula is proved unsatisfiable without checking for worlds where $\neg B$ holds. At the same time, if one can show unsatisfiability of $\neg B$, there is no need to check $\neg A$. As a result, while there are two possible way to expand $\neg\Box A \wedge \neg\Box B$, one of these two ways is always sufficient to prove unsatisfiability if the formula is unsatisfiable. For example, one may expand the tableau by considering an arbitrary world where $\neg A$ holds. If this expansion leads to unsatisfiability, the original formula is unsatisfiable. However, it is also possible that unsatisfiability cannot be proved this way, and that the world where $\neg B$ holds should have been considered instead. As a result, one can always prove unsatisfiability by expanding either $\neg\Box A$ only or $\neg\Box B$ only; however, if the wrong choice is done the resulting tableau may not be closed. Expanding either subformula leads to tableau calculi that are complete but not proof-confluent. Searching as described in the "Searching for a closed tableau" may therefore be necessary.

Depending on whether the precondition and consequence of a tableau expansion rule refer to the same world or not, the rule is called static or transactional. While rules for propositional connectives are all static, not all rules for modal connectives are transactional: for example, in every modal logic including axiom T, it holds that $\Box A$ implies A in the same world. As a result, the relative (modal) tableau expansion rule is static, as both its precondition and consequence refer to the same world.

Formula-deleting tableau

A way for making formulae referring to different worlds not interacting in the wrong way is to make sure that all formulae of a branch refer to the same world. This condition is initially true as all formulae in the set to be checked for consistency are assumed referring to the same world. When expanding a branch, two situations are possible: either the new formulae refer to the same world as the other one in the branch or not. In the first case, the rule is applied normally. In the second case, all formulae of the branch that do not also hold in the new world are deleted from the branch, and possibly added to all other branches that are still relative to the old world.

As an example, in S5 every formula $\Box A$ that is true in a world is also true in all accessible worlds (that is, in all accessible worlds both A and $\Box A$ are true). Therefore, when applying $\frac{\neg\Box A}{\neg A}$, whose consequence holds in a different world, one deletes all formulae from the branch, but can keep all formulae $\Box A$, as these hold in the new world as well. In order to retain completeness, the deleted formulae are then added to all other branches that still refer to the old world.

World-labeled tableau

A different mechanism for ensuring the correct interaction between formulae referring to different worlds is to switch from formulae to labeled formulae: instead of writing A , one would write $w : A$ to make it explicit that A holds in world w .

All propositional expansion rules are adapted to this variant by stating that they all refer to formulae with the same world label. For example, $w : A \wedge B$ generates two nodes labeled with $w : A$ and $w : B$; a branch is closed only if it contains two opposite literals of the same world, like $w : a$ and $w : \neg a$; no closure is generated if the two world labels are different, like in $w : a$ and $w' : \neg a$.

The modal expansion rule may have a consequence that refer to a different worlds. For example, the rule for $\neg\Box A$ would be written as follows

$$\frac{w : \neg\Box A}{w' : \neg A}$$

The precondition and consequent of this rule refer to worlds w and w' , respectively. The various different calculi use different methods for keeping track of the accessibility of the worlds used as labels. Some include pseudo-formulae like wRw' to denote that w' is accessible from w . Some others use sequences of integers as world labels, this notation implicitly representing the accessibility relation (for example, $(1, 4, 2, 3)$ is accessible from $(1, 4, 2)$.)

Set-labeling tableaux

The problem of interaction between formulae holding in different worlds can be overcome by using set-labeling tableaux. These are trees whose nodes are labeled with sets of formulae; the expansion rules tell how to attach new nodes to a leaf, based only on the label of the leaf (and not on the label of other nodes in the branch).

Tableaux for modal logics are used to verify the satisfiability of a set of modal formulae in a given modal logic. Given a set of formulae S , they check the existence of a model M and a world w such that $M, w \models S$.

The expansion rules depend on the particular modal logic used. A tableau system for the basic modal logic K can be obtained by adding to the propositional tableau rules the following one:

$$(K) \frac{\Box A_1; \dots; \Box A_n; \neg\Box B}{A_1; \dots; A_n; \neg B}$$

Intuitively, the precondition of this rule expresses the truth of all formulae A_1, \dots, A_n at all accessible worlds, and truth of $\neg B$ at some accessible worlds. The consequence of this rule is a formula that must be true at one of those worlds where $\neg B$ is true.

More technically, modal tableaux methods check the existence of a model M and a world w that make set of formulae true. If $\Box A_1; \dots; \Box A_n; \neg \Box B$ are true in w , there must be a world w' that is accessible from w and that makes $A_1; \dots; A_n; \neg B$ true. This rule therefore amounts to deriving a set of formulae that must be satisfied in such w' .

While the preconditions $\Box A_1; \dots; \Box A_n; \neg \Box B$ are assumed satisfied by M, w , the consequences $A_1; \dots; A_n; \neg B$ are assumed satisfied in M, w' : same model but possibly different worlds. Set-labeled tableaux do not explicitly keep track of the world where each formula is assumed true: two nodes may or may not refer to the same world. However, the formulae labeling any given node are assumed true at the same world. As a result of the possibly different worlds where formulae are assumed true, a formula in a node is not automatically valid in all its descendants, as every application of the modal rule correspond to a move from a world to another one. This condition is automatically captured by set-labeling tableaux, as expansion rules are based only on the leaf where they are applied and not on its ancestors.

Remarkably, (K) does not directly extend to multiple negated boxed formulae such as in $\Box A_1; \dots; \Box A_n; \neg \Box B_1; \neg \Box B_2$: while there exists an accessible world where B_1 is false and one in which B_2 is false, these two worlds are not necessarily the same.

Differently from the propositional rules, (K) states conditions over all its preconditions. For example, it cannot be applied to a node labeled by $a; \Box b; \Box(b \rightarrow c); \neg \Box c$; while this set is inconsistent and this could be easily proved by applying (K) , this rule cannot be applied because of formula a , which is not even relevant to inconsistency. Removal of such formulae is made possible by the rule:

$$(\theta) \frac{A_1; \dots; A_n; B_1; \dots; B_m}{A_1; \dots; A_n}$$

The addition of this rule (thinning rule) makes the resulting calculus non-confluent: a tableau for an inconsistent set may be impossible to close, even if a closed tableau for the same set exists.

Rule (θ) is non-deterministic: the set of formulae to be removed (or to be kept) can be chosen arbitrarily; this creates the problem of choosing a set of formulae to discard that is not so large it makes the resulting set satisfiable and not so small it makes the necessary expansion rules inapplicable. Having a large number of possible choices makes the problem of searching for a closed tableau harder.

This non-determinism can be avoided by restricting the usage of (θ) so that it is only applied before a modal expansion rule, and so that it only removes the formulae that make that other rule inapplicable. This condition can be also formulated by merging the two rules in a single one. The resulting rule produces the same result as the old one, but implicitly discard all formulae that made the old rule inapplicable. This mechanism for removing (θ) has been proved to preserve completeness for many modal logics.

Axiom T expresses reflexivity of the accessibility relation: every world is accessible from itself. The corresponding tableau expansion rule is:

$$(T) \frac{A_1; \dots; A_n; \Box B}{A_1; \dots; A_n; \Box B; B}$$

This rule relates conditions over the same world: if $\Box B$ is true in a world, by reflexivity B is also true *in the same world*. This rule is static, not transactional, as both its precondition and consequent refer to the same world.

This rule copies $\Box B$ from the precondition to the consequent, in spite of this formula having being "used" to generate B . This is correct, as the considered world is the same, so $\Box B$ also holds there. This "copying" is necessary in some cases. It is for example necessary to prove the inconsistency of $\Box(a \wedge \neg \Box a)$: the only applicable rules are in order $(T), (\wedge), (\theta), (K)$, from which one is blocked if $\Box a$ is not copied.

Auxiliary tableaux

A different method for dealing with formulae holding in alternate worlds is to start a different tableau for each new world that is introduced in the tableau. For example, $\neg\Box A$ implies that A is false in an accessible world, so one starts a new tableau rooted by $\neg A$. This new tableau is attached to the node of the original tableau where the expansion rule has been applied; a closure of this tableau immediately generates a closure of all branches where that node is, regardless of whether the same node is associated other auxiliary tableaux. The expansion rules for the auxiliary tableaux are the same as for the original one; therefore, an auxiliary tableau can have in turns other (sub-)auxiliary tableaux.

Global assumptions

The above modal tableaux establish the consistency of a set of formulae, and can be used for solving the local logical consequence problem. This is the problem of telling whether, for each model M , if A is true in a world w , then B is also true in the same world. This is the same as checking whether B is true in a world of a model, in the assumption that A is also true in the same world of the same model.

A related problem is the global consequence problem, where the assumption is that a formula (or set of formulae) G is true in all possible worlds of the model. The problem is that of checking whether, in all models M where G is true in all worlds, B is also true in all worlds.

Local and global assumption differ on models where the assumed formula is true in some worlds but not in others. As an example, $\{P, \neg\Box(P \wedge Q)\}$ entails $\neg\Box Q$ globally but not locally. Local entailment does not hold in a model consisting of two worlds making P and $\neg P, Q$ true, respectively, and where the second is accessible from the first; in the first world, the assumption is true but $\Box Q$ is false. This counterexample works because P can be assumed true in a world and false in another one. If however the same assumption is considered global, $\neg P$ is not allowed in any world of the model.

These two problems can be combined, so that one can check whether B is a local consequence of A under the global assumption G . Tableaux calculi can deal with global assumption by a rule allowing its addition to every node, regardless of the world it refers to.

Notations

The following conventions are sometimes used.

Uniform notation

When writing tableaux expansion rules, formulae are often denoted using a convention, so that for example α is always considered to be $\alpha_1 \wedge \alpha_2$. The following table provides the notation for formulae in propositional, first-order, and modal logic.

| Notation | Formulae | | |
|----------|----------------------------|--|--|
| α | $\alpha_1 \wedge \alpha_2$ | $\neg(\overline{\alpha_1} \vee \overline{\alpha_2})$ | $\neg(\alpha_1 \rightarrow \overline{\alpha_2})$ |
| β | $\beta_1 \vee \beta_2$ | $\overline{\beta_1} \rightarrow \beta_2$ | $\neg(\overline{\beta_1} \wedge \overline{\beta_2})$ |
| γ | $\forall x \gamma_1(x)$ | $\neg \exists x \overline{\gamma_1(x)}$ | |
| δ | $\exists x \delta_1(x)$ | $\neg \forall x \overline{\delta_1(x)}$ | |
| π | $\Box \pi_1$ | $\neg \Diamond \overline{\pi_1}$ | |
| ν | $\Diamond \nu_1$ | $\neg \Box \overline{\nu_1}$ | |

Each label in the first column is taken to be either formula in the other columns. An overlined formula such as $\overline{\alpha_1}$ indicates that α_1 is the negation of whatever formula appears in its place, so that for example in formula $\neg(a \vee b)$ the subformula α_1 is the negation of a .

Since every label indicates many equivalent formulae, this notation allows writing a single rule for all these equivalent formulae. For example, the conjunction expansion rule is formulated as:

$$(\alpha) \frac{\alpha}{\alpha_1 \quad \alpha_2}$$

Signed formulae

A formula in a tableau is assumed true. Signed tableaux allows stating that a formula is false. This is generally achieved by adding a label to each formula, where the label **T** indicates formulae assumed true and **F** those assumed false. A different but equivalent notation is that to write formulae that are assumed true at the left of the node and formulae assumed false at its right.

References

- Bostock, David, 1997. *Intermediate Logic*. Oxford Univ. Press.
- M D'Agostino, D Gabbay, R Haehnle, J Posegga (Eds), *Handbook of Tableau Methods*, Kluwer, 1999.
- Girle, Rod, 2000. *Modal Logics and Philosophy*. Teddington UK: Acumen.
- Goré, Rajeev (1999) "Tableau Methods for Modal and Temporal Logics" in D'Agostino, M., Dov Gabbay, R. Haehnle, and J. Posegga, eds., *Handbook of Tableau Methods*. Kluwer: 297-396.
- Richard Jeffrey, 1990 (1967). *Formal Logic: Its Scope and Limits*, 3rd ed. McGraw Hill.
- Raymond Smullyan, 1995 (1968). *First Order-Logic*. Dover Publications.
- Melvin Fitting (1996). *First-order logic and automated theorem proving* (2nd ed.). Springer-Verlag.
- Reiner Hähnle (2001). *Tableaux and Related Methods*. Handbook of Automated Reasoning
- Reinhold Letz, Gernot Stenz (2001). *Model Elimination and Connection Tableau Procedures*. Handbook of Automated Reasoning
- Zeman, J. J. (1973) *Modal Logic*. ^[1] Reidel.

External links

- TABLEAUX ^[2]: an annual international conference on automated reasoning with analytic tableaux and related methods.
- JAR ^[3]: Journal of Automated Reasoning.
- lolo ^[4]: a simple theorem prover written in Haskell that uses analytic tableaux for propositional logic.
- tableaux.cgi ^[5]: an interactive prover for propositional and first-order logic using tableaux.

References

- [1] <http://www.clas.ufl.edu/users/jzeman/modallogic/>
- [2] <http://i12www.ira.uka.de/TABLEAUX/>
- [3] <http://www-unix.mcs.anl.gov/JAR/>
- [4] <http://lolo.svn.sourceforge.net/viewvc/lolo/>
- [5] <http://www.ncc.up.pt/~pbv/cgi/tableaux.cgi>

Natural deduction

In logic and proof theory, **natural deduction** is a kind of proof calculus in which logical reasoning is expressed by inference rules closely related to the "natural" way of reasoning. This contrasts with the axiomatic systems which instead use axioms as much as possible to express the logical laws of deductive reasoning.

Motivation

Natural deduction grew out of a context of dissatisfaction with the axiomatizations of deductive reasoning common to the systems of Hilbert, Frege, and Russell (see, e.g., Hilbert system). Such axiomatizations were most famously used by Russell and Whitehead in their mathematical treatise *Principia Mathematica*. Spurred on by a series of seminars in Poland in 1926 by Łukasiewicz that advocated a more natural treatment of logic, Jaśkowski made the earliest attempts at defining a more natural deduction, first in 1929 using a diagrammatic notation, and later updating his proposal in a sequence of papers in 1934 and 1935. His proposals led to different notations such as Fitch-style calculus (or Fitch's diagrams) or Suppes' method of which e.g. Lemmon gave a variant called system L.

Natural deduction in its modern form was independently proposed by the German mathematician Gentzen in 1935, in a dissertation delivered to the faculty of mathematical sciences of the university of Göttingen. The term *natural deduction* (or rather, its German equivalent *natürliches Schließen*) was coined in that paper:

Ich wollte zunächst einmal einen Formalismus aufstellen, der dem wirklichen Schließen möglichst nahe kommt. So ergab sich ein „Kalkül des natürlichen Schließens“.

(First I wished to construct a formalism that comes as close as possible to actual reasoning. Thus arose a "calculus of natural deduction".)

—Gentzen, *Untersuchungen über das logische Schließen* (Mathematische Zeitschrift 39, pp.176–210, 1935)

Gentzen was motivated by a desire to establish the consistency of number theory, and he found immediate use for his natural deduction calculus. He was nevertheless dissatisfied with the complexity of his proofs, and in 1938 gave a new consistency proof using his sequent calculus. In a series of seminars in 1961 and 1962 Prawitz gave a comprehensive summary of natural deduction calculi, and transported much of Gentzen's work with sequent calculi into the natural deduction framework. His 1965 monograph *Natural deduction: a proof-theoretical study* was to become a reference work on natural deduction, and included applications for modal and second-order logic.

In natural deduction, a proposition is deduced from a collection of premises by applying inference rules repeatedly. The system presented in this article is a minor variation of Gentzen's or Prawitz's formulation, but with a closer adherence to Martin-Löf's description of logical judgments and connectives (Martin-Löf, 1996).

Judgments and propositions

A *judgment* is something that is knowable, that is, an object of knowledge. It is *evident* if one in fact knows it. Thus "it is raining" is a judgment, which is evident for the one who knows that it is actually raining; in this case one may readily find evidence for the judgment by looking outside the window or stepping out of the house. In mathematical logic however, evidence is often not as directly observable, but rather deduced from more basic evident judgments. The process of deduction is what constitutes a *proof*; in other words, a judgment is evident if one has a proof for it.

The most important judgments in logic are of the form "*A is true*". The letter *A* stands for any expression representing a *proposition*; the truth judgments thus require a more primitive judgment: "*A is a proposition*". Many other judgments have been studied; for example, "*A is false*" (see classical logic), "*A is true at time t*" (see temporal logic), "*A is necessarily true*" or "*A is possibly true*" (see modal logic), "*the program M has type τ*" (see programming languages and type theory), "*A is achievable from the available resources*" (see linear logic), and many others. To start with, we shall concern ourselves with the simplest two judgments "*A is a proposition*" and "*A is true*", abbreviated as "*A prop*" and "*A true*" respectively.

The judgment "*A prop*" defines the structure of valid proofs of *A*, which in turn defines the structure of propositions. For this reason, the inference rules for this judgment are sometimes known as *formation rules*. To illustrate, if we have two propositions *A* and *B* (that is, the judgments "*A prop*" and "*B prop*" are evident), then we form the compound proposition *A and B*, written symbolically as "*A ∧ B*". We can write this in the form of an inference rule:

$$\frac{A \text{ prop} \quad B \text{ prop}}{A \wedge B \text{ prop}} \wedge_F$$

This inference rule is *schematic*: *A* and *B* can be instantiated with any expression. The general form of an inference rule is:

$$\frac{J_1 \quad J_2 \quad \dots \quad J_n}{J} \text{ name}$$

where each J_i is a judgment and the inference rule is named "name". The judgments above the line are known as *premises*, and those below the line are *conclusions*. Other common logical propositions are disjunction ($A \vee B$), negation ($\neg A$), implication ($A \supset B$), and the logical constants truth (\top) and falsehood (\perp). Their formation rules are below.

$$\frac{A \text{ prop} \quad B \text{ prop}}{A \vee B \text{ prop}} \vee_F \quad \frac{A \text{ prop} \quad B \text{ prop}}{A \supset B \text{ prop}} \supset_F \quad \frac{}{\top \text{ prop}} \top_F \quad \frac{}{\perp \text{ prop}} \perp_F \quad \frac{A \text{ prop}}{\neg A \text{ prop}} \neg_F$$

Introduction and elimination

Now we discuss the "*A true*" judgment. Inference rules that introduce a logical connective in the conclusion are known as *introduction rules*. To introduce conjunctions, *i.e.*, to conclude "*A and B true*" for propositions *A* and *B*, one requires evidence for "*A true*" and "*B true*". As an inference rule:

$$\frac{A \text{ true} \quad B \text{ true}}{A \wedge B \text{ true}} \wedge_I$$

It must be understood that in such rules the objects are propositions. That is, the above rule is really an abbreviation for:

$$\frac{A \text{ prop} \quad B \text{ prop} \quad A \text{ true} \quad B \text{ true}}{A \wedge B \text{ true}} \wedge_I$$

This can also be written:

$$\frac{A \wedge B \text{ prop} \quad A \text{ true} \quad B \text{ true}}{A \wedge B \text{ true}} \wedge_I$$

In this form, the first premise can be satisfied by the \wedge_I formation rule, giving the first two premises of the previous form. In this article we shall elide the "prop" judgments where they are understood. In the nullary case, one can derive truth from no premises.

$$\frac{}{\top \text{ true}} \top_I$$

If the truth of a proposition can be established in more than one way, the corresponding connective has multiple introduction rules.

$$\frac{A \text{ true}}{A \vee B \text{ true}} \vee_{I1} \quad \frac{B \text{ true}}{A \vee B \text{ true}} \vee_{I2}$$

Note that in the nullary case, *i.e.*, for falsehood, there are *no* introduction rules. Thus one can never infer falsehood from simpler judgments.

Dual to introduction rules are *elimination rules* to describe how to de-construct information about a compound proposition into information about its constituents. Thus, from " $A \sqcap B \text{ true}$ ", we can conclude " $A \text{ true}$ " and " $B \text{ true}$ ":

$$\frac{A \wedge B \text{ true}}{A \text{ true}} \wedge_{E1} \quad \frac{A \wedge B \text{ true}}{B \text{ true}} \wedge_{E2}$$

As an example of the use of inference rules, consider commutativity of conjunction. If $A \sqcap B$ is true, then $B \sqcap A$ is true; This derivation can be drawn by composing inference rules in such a fashion that premises of a lower inference match the conclusion of the next higher inference.

$$\frac{\frac{A \wedge B \text{ true}}{B \text{ true}} \wedge_{E2} \quad \frac{A \wedge B \text{ true}}{A \text{ true}} \wedge_{E1}}{B \wedge A \text{ true}} \wedge_I$$

The inference figures we have seen so far are not sufficient to state the rules of implication introduction or disjunction elimination; for these, we need a more general notion of *hypothetical derivation*.

Hypothetical derivations

A pervasive operation in mathematical logic is *reasoning from assumptions*. For example, consider the following derivation:

$$\frac{A \wedge (B \wedge C) \text{ true}}{B \wedge C \text{ true}} \wedge E_2$$

$$\frac{B \wedge C \text{ true}}{B \text{ true}} \wedge E_1$$

This derivation does not establish the truth of B as such; rather, it establishes the following fact:

If $A \sqcap (B \sqcap C)$ is true then B is true.

In logic, one says "*assuming $A \sqcap (B \sqcap C)$ is true, we show that B is true*"; in other words, the judgement " $B \text{ true}$ " depends on the assumed judgement " $A \sqcap (B \sqcap C) \text{ true}$ ". This is a *hypothetical derivation*, which we write as follows:

$$\frac{A \wedge (B \wedge C) \text{ true}}{\vdots} \quad B \text{ true}$$

The interpretation is: " $B \text{ true}$ is derivable from $A \sqcap (B \sqcap C) \text{ true}$ ". Of course, in this specific example we actually know the derivation of " $B \text{ true}$ " from " $A \sqcap (B \sqcap C) \text{ true}$ ", but in general we may not *a-priori* know the derivation. The general form of a hypothetical derivation is:

$$\frac{D_1 \quad D_2 \cdots D_n}{J}$$

Each hypothetical derivation has a collection of *antecedent* derivations (the D_i) written on the top line, and a *succedent* judgement (J) written on the bottom line. Each of the premises may itself be a hypothetical derivation.

(For simplicity, we treat a judgement as a premise-less derivation.)

The notion of hypothetical judgement is *internalised* as the connective of implication. The introduction and elimination rules are as follows.

$$\frac{\overline{A \text{ true}}^u}{\vdots} \quad \frac{B \text{ true}}{A \supset B \text{ true}} \supset I^u \quad \frac{A \supset B \text{ true} \quad A \text{ true}}{B \text{ true}} \supset E$$

In the introduction rule, the antecedent named u is *discharged* in the conclusion. This is a mechanism for delimiting the *scope* of the hypothesis: its sole reason for existence is to establish " $B \text{ true}$ "; it cannot be used for any other purpose, and in particular, it cannot be used below the introduction. As an example, consider the derivation of " $A \sqcap (B \sqcap (A \sqcup B)) \text{ true}$ ":

$$\frac{\frac{\overline{A \text{ true}}^u \quad \overline{B \text{ true}}^w}{A \wedge B \text{ true}} \wedge I}{B \supset (A \wedge B) \text{ true}} \supset I^w \quad \frac{B \supset (A \wedge B) \text{ true}}{A \supset (B \supset (A \wedge B)) \text{ true}} \supset I^u$$

This full derivation has no unsatisfied premises; however, sub-derivations *are* hypothetical. For instance, the derivation of " $B \sqcap (A \sqcup B) \text{ true}$ " is hypothetical with antecedent " $A \text{ true}$ " (named u).

With hypothetical derivations, we can now write the elimination rule for disjunction:

$$\frac{A \vee B \text{ true} \quad \frac{\overline{A \text{ true}}^u \quad \overline{B \text{ true}}^w}{\vdots} \quad \frac{C \text{ true} \quad C \text{ true}}{C \text{ true}} \vee E^{u,w}}{C \text{ true}}$$

In words, if $A \sqcup B$ is true, and we can derive $C \text{ true}$ both from $A \text{ true}$ and from $B \text{ true}$, then C is indeed true. Note that this rule does not commit to either $A \text{ true}$ or $B \text{ true}$. In the zero-ary case, *i.e.* for falsehood, we obtain the following elimination rule:

$$\frac{\perp \text{ true}}{C \text{ true}} \perp E$$

This is read as: if falsehood is true, then any proposition C is true.

Negation is similar to implication.

$$\frac{\overline{A \text{ true}}^u}{\vdots} \quad \frac{p \text{ true}}{\neg A \text{ true}} \neg I^{u,p} \quad \frac{\neg A \text{ true} \quad A \text{ true}}{C \text{ true}} \neg E$$

The introduction rule discharges both the name of the hypothesis u , and the succedent p , *i.e.*, the proposition p must not occur in the conclusion A . Since these rules are schematic, the interpretation of the introduction rule is: if from " $A \text{ true}$ " we can derive for every proposition p that " $p \text{ true}$ ", then A must be false, *i.e.*, " $\text{not } A \text{ true}$ ". For the elimination, if both A and $\text{not } A$ are shown to be true, then there is a contradiction, in which case every proposition C is true. Because the rules for implication and negation are so similar, it should be fairly easy to see that $\text{not } A$ and $A \sqcap$ are equivalent, *i.e.*, each is derivable from the other.

Consistency, completeness, and normal forms

A theory is said to be consistent if falsehood is not provable (from no assumptions) and is complete if every theorem is provable using the inference rules of the logic. These are statements about the entire logic, and are usually tied to some notion of a model. However, there are local notions of consistency and completeness that are purely syntactic checks on the inference rules, and require no appeals to models. The first of these is local consistency, also known as local reducibility, which says that any derivation containing an introduction of a connective followed immediately by its elimination can be turned into an equivalent derivation without this detour. It is a check on the *strength* of elimination rules: they must not be so strong that they include knowledge not already contained in its premises. As an example, consider conjunctions.

$$\begin{array}{c}
 \text{----- } u \quad \text{----- } w \\
 A \text{ true} \quad B \text{ true} \\
 \text{----- } \wedge I \\
 A \wedge B \text{ true} \\
 \text{----- } \wedge E_1 \\
 A \text{ true}
 \end{array}
 \Rightarrow
 \begin{array}{c}
 \text{----- } u \\
 A \text{ true}
 \end{array}$$

Dually, local completeness says that the elimination rules are strong enough to decompose a connective into the forms suitable for its introduction rule. Again for conjunctions:

$$\begin{array}{c}
 \text{----- } u \\
 A \wedge B \text{ true}
 \end{array}
 \Rightarrow
 \begin{array}{c}
 \text{----- } u \quad \text{----- } u \\
 A \wedge B \text{ true} \quad A \wedge B \text{ true} \\
 \text{----- } \wedge E_1 \quad \text{----- } \wedge E_2 \\
 A \text{ true} \quad B \text{ true} \\
 \text{----- } \wedge I \\
 A \wedge B \text{ true}
 \end{array}$$

These notions correspond exactly to β -reduction (beta reduction) and η -conversion (eta conversion) in the lambda calculus, using the Curry-Howard isomorphism. By local completeness, we see that every derivation can be converted to an equivalent derivation where the principal connective is introduced. In fact, if the entire derivation obeys this ordering of eliminations followed by introductions, then it is said to be *normal*. In a normal derivation all eliminations happen above introductions. In most logics, every derivation has an equivalent normal derivation, called a *normal form*. The existence of normal forms is generally hard to prove using natural deduction alone, though such accounts do exist in the literature, most notably by Dag Prawitz in 1961; see his book *Natural deduction: a proof-theoretical study*, A&W Stockholm 1965, no ISBN. It is much easier to show this indirectly by means of a cut-free sequent calculus presentation.

First and higher-order extensions

The logic of the earlier section is an example of a *single-sorted* logic, *i.e.*, a logic with a single kind of object: propositions. Many extensions of this simple framework have been proposed; in this section we will extend it with a second sort of *individuals* or *terms*. More precisely, we will add a new kind of judgement, "*t is a term*" (or "*t term*") where *t* is schematic. We shall fix a countable set *V* of *variables*, another countable set *F* of *function symbols*, and construct terms as follows:

| | |
|---|--|
| contexts | |
| Σ | term variables |
| Γ | true hypotheses |
| terms | |
| $\frac{}{\Sigma, v \vdash v \text{ term}} \text{var-F}$ | $\frac{f \in F \quad \Sigma \vdash t_1 \text{ term} \quad \Sigma \vdash t_2 \text{ term} \quad \dots \quad \Sigma \vdash t_n \text{ term}}{\Sigma \vdash f(t_1, t_2, \dots, t_n) \text{ term}} \text{app-F}$ |
| propositions | |
| $\frac{\varphi \in P \quad \Sigma \vdash t_1 \text{ term} \quad \Sigma \vdash t_2 \text{ term} \quad \dots \quad \Sigma \vdash t_n \text{ term}}{\Sigma \vdash \varphi(t_1, t_2, \dots, t_n) \text{ prop}} \text{pred-F}$ | |
| $\frac{\Sigma \vdash A \text{ prop} \quad \Sigma \vdash B \text{ prop}}{\Sigma \vdash A \wedge B \text{ prop}} \wedge F$ | $\frac{\Sigma \vdash A \text{ prop} \quad \Sigma \vdash B \text{ prop}}{\Sigma \vdash A \vee B \text{ prop}} \vee F$ |
| $\frac{\Sigma \vdash A \text{ prop} \quad \Sigma \vdash B \text{ prop}}{\Sigma \vdash A \supset B \text{ prop}} \supset F$ | $\frac{\Sigma, x \vdash A \text{ prop}}{\Sigma \vdash \forall x. A \text{ prop}} \forall F$ |
| $\frac{\Sigma, x \vdash A \text{ prop}}{\Sigma \vdash \exists x. A \text{ prop}} \exists F$ | |
| judgemental rules | |
| $\frac{}{\Sigma \vdash \Gamma, u : A \vdash u : A} \text{hyp}$ | |
| introduction rules | |
| $\frac{\Sigma \vdash \Gamma \vdash s_1 : A \quad \Sigma \vdash \Gamma \vdash s_2 : B}{\Sigma \vdash \Gamma \vdash (s_1, s_2) : A \wedge B} \wedge I$ | $\frac{}{\Sigma \vdash \Gamma \vdash () : \top} \top I$ |
| $\frac{\Sigma \vdash \Gamma \vdash \pi : A}{\Sigma \vdash \Gamma \vdash \text{fst } \pi : A \vee B} \vee I_1$ | $\frac{\Sigma \vdash \Gamma \vdash \pi : B}{\Sigma \vdash \Gamma \vdash \text{snd } \pi : A \vee B} \vee I_2$ |
| $\frac{\Sigma \vdash \Gamma, u : A \vdash \pi : B}{\Sigma \vdash \Gamma \vdash \text{let } u : A \supset B} \supset I$ | $\frac{\Sigma, x \vdash \Gamma \vdash \pi : A}{\Sigma \vdash \Gamma \vdash \lambda x. \pi : \lambda x. A} \lambda I$ |
| $\frac{\Sigma \vdash \Gamma \vdash \pi : A \wedge B}{\Sigma \vdash \Gamma \vdash \text{fst } \pi : A} \wedge E_1$ | $\frac{\Sigma \vdash \Gamma \vdash \pi : A \wedge B}{\Sigma \vdash \Gamma \vdash \text{snd } \pi : B} \wedge E_2$ |
| $\frac{\Sigma \vdash \Gamma \vdash \pi : A \vee B \quad \Sigma \vdash \Gamma, u : A \vdash s_1 : C \quad \Sigma \vdash \Gamma, u : B \vdash s_2 : C}{\Sigma \vdash \Gamma \vdash \text{case } \pi \text{ of } \text{fst } u \Rightarrow s_1 \mid \text{snd } u \Rightarrow s_2 : C} \vee E$ | $\frac{\Sigma \vdash \Gamma \vdash \pi : \perp}{\Sigma \vdash \Gamma \vdash \text{abort } \pi : C} \perp E$ |
| $\frac{\Sigma \vdash \Gamma \vdash s_1 : A \supset B \quad \Sigma \vdash \Gamma \vdash s_2 : A}{\Sigma \vdash \Gamma \vdash s_1 s_2 : B} \supset E$ | $\frac{\Sigma \vdash \Gamma \vdash \pi : \lambda x. A \quad \Sigma \vdash \Gamma \vdash a : A}{\Sigma \vdash \Gamma \vdash a[\pi] : A} \lambda E$ |
| $\frac{\Sigma \vdash \Gamma \vdash s_1 : \exists x. A \quad \Sigma, x \vdash \Gamma, u : A \vdash s_2 : C}{\Sigma \vdash \Gamma \vdash \text{let } (x, u) = s_1 \text{ in } s_2 : C} \exists E$ | |

Summary of first-order system

| | | | | | |
|------------------|-----------|--------------------|--------------------|---------|--|
| $v \in V$ | $f \in F$ | $t_1 \text{ term}$ | $t_2 \text{ term}$ | \dots | $t_n \text{ term}$ |
| ----- var-F | | | | | ----- app-F |
| $v \text{ term}$ | | | | | $f(t_1, t_2, \dots, t_n) \text{ term}$ |

For propositions, we consider a third countable set *P* of *predicates*, and define *atomic predicates over terms* with the following formation rule:

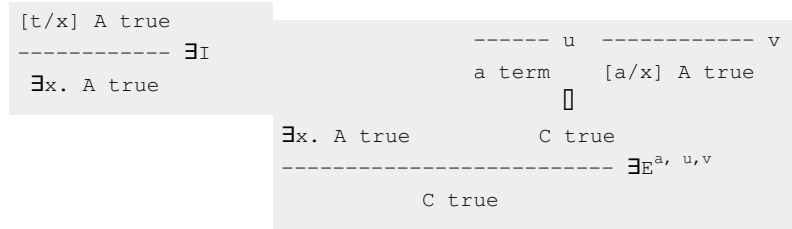
| | | | | |
|-----------------|--|--------------------|---------|--------------------|
| $\varphi \in P$ | $t_1 \text{ term}$ | $t_2 \text{ term}$ | \dots | $t_n \text{ term}$ |
| ----- | | | | ----- pred-F |
| | $\varphi(t_1, t_2, \dots, t_n) \text{ prop}$ | | | |

In addition, we add a pair of *quantified* propositions: universal (\forall) and existential (\exists):

| | |
|-----------------------------|-----------------------------|
| ----- u | ----- u |
| x term | x term |
| \square | \square |
| A prop | A prop |
| ----- \forall_F^u | ----- \exists_F^u |
| $\forall x. A \text{ prop}$ | $\exists x. A \text{ prop}$ |

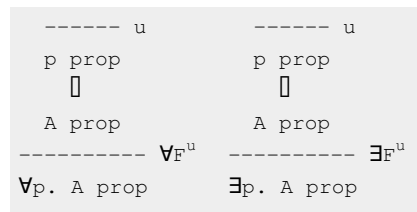
These quantified propositions have the following introduction and elimination rules.

| | |
|-----------------------------|---|
| ----- u | $\forall x. A \text{ true } t \text{ term}$ |
| a term | ----- $\forall E$ |
| \square | $[t/x] A \text{ true}$ |
| $[a/x] A \text{ true}$ | |
| ----- $\forall I^u, a$ | |
| $\forall x. A \text{ true}$ | |



In these rules, the notation $[t/x] A$ stands for the substitution of t for every (visible) instance of x in A , avoiding capture; see the article on lambda calculus for more detail about this standard operation. As before the superscripts on the name stand for the components that are discharged: the term a cannot occur in the conclusion of $\forall I$ (such terms are known as *eigenvariables* or *parameters*), and the hypotheses named u and v in $\exists E$ are localised to the second premise in a hypothetical derivation. Although the propositional logic of earlier sections was decidable, adding the quantifiers makes the logic undecidable.

So far the quantified extensions are *first-order*: they distinguish propositions from the kinds of objects quantified over. Higher-order logic takes a different approach and has only a single sort of propositions. The quantifiers have as the domain of quantification the very same sort of propositions, as reflected in the formation rules:



A discussion of the introduction and elimination forms for higher-order logic is beyond the scope of this article. It is possible to be in between first-order and higher-order logics. For example, second-order logic has two kinds of propositions, one kind quantifying over terms, and the second kind quantifying over propositions of the first kind.

Different presentations of natural deduction

Tree-like presentations

Gentzen's discharging annotations used to internalise hypothetical judgment can be avoided by representing proofs as a tree of sequents $\Gamma \multimap A$ instead of a tree of $A \text{ true}$ judgments.

Sequential presentations

Jaśkowski's representations of natural deduction led to different notations such as Fitch-style calculus (or Fitch's diagrams) or Suppes' method of which e.g. Lemmon gave a variant called system L.

Proofs and type-theory

The presentation of natural deduction so far has concentrated on the nature of propositions without giving a formal definition of a *proof*. To formalise the notion of proof, we alter the presentation of hypothetical derivations slightly. We label the antecedents with *proof variables* (from some countable set V of variables), and decorate the succedent with the actual proof. The antecedents or *hypotheses* are separated from the succedent by means of a *turnstile* (\vdash). This modification sometimes goes under the name of *localised hypotheses*. The following diagram summarises the change.

A ". The logical connectives are also given a different reading: conjunction is viewed as product (\times), implication as the function arrow (\rightarrow), etc. The differences are only cosmetic, however. Type theory has a natural deduction presentation in terms of formation, introduction and elimination rules; in fact, the reader can easily reconstruct what is known as *simple type theory* from the previous sections.

The difference between logic and type theory is primarily a shift of focus from the types (propositions) to the programs (proofs). Type theory is chiefly interested in the convertibility or reducibility of programs. For every type, there are canonical programs of that type which are irreducible; these are known as *canonical forms* or *values*. If every program can be reduced to a canonical form, then the type theory is said to be *normalising* (or *weakly normalising*). If the canonical form is unique, then the theory is said to be *strongly normalising*. Normalisability is a rare feature of most non-trivial type theories, which is a big departure from the logical world. (Recall that every logical derivation has an equivalent normal derivation.) To sketch the reason: in type theories that admit recursive definitions, it is possible to write programs that never reduce to a value; such looping programs can generally be given any type. In particular, the looping program has type \perp , although there is no logical proof of " \perp true". For this reason, the *propositions as types; proofs as programs* paradigm only works in one direction, if at all: interpreting a type theory as a logic generally gives an inconsistent logic.

Like logic, type theory has many extensions and variants, including first-order and higher-order versions. An interesting branch of type theory, known as dependent type theory, allows quantifiers to range over programs themselves. These quantified types are written as Π and Σ instead of \forall and \exists , and have the following formation rules:

| | | | |
|---|-------------------------------------|--|-------------------------------------|
| $\Gamma \vdash A \text{ type}$ | $\Gamma, x:A \vdash B \text{ type}$ | $\Gamma \vdash A \text{ type}$ | $\Gamma, x:A \vdash B \text{ type}$ |
| ----- Π -F | | ----- Σ -F | |
| $\Gamma \vdash \Pi x:A. B \text{ type}$ | | $\Gamma \vdash \Sigma x:A. B \text{ type}$ | |

These types are generalisations of the arrow and product types, respectively, as witnessed by their introduction and elimination rules.

| | | |
|---|---|---------------------------|
| $\Gamma, x:A \vdash \pi : B$ | $\Gamma \vdash \pi_1 : \Pi x:A. B$ | $\Gamma \vdash \pi_2 : A$ |
| ----- Π I | ----- Π E | |
| $\Gamma \vdash \lambda x. \pi : \Pi x:A. B$ | $\Gamma \vdash \pi_1 \pi_2 : [\pi_2/x] B$ | |

| | | | |
|--|--------------------------------|--------------------------------------|---|
| $\Gamma \vdash \pi_1 : A$ | $\Gamma, x:A \vdash \pi_2 : B$ | $\Gamma \vdash \pi : \Sigma x:A. B$ | $\Gamma \vdash \pi : \Sigma x:A. B$ |
| ----- Σ I | | ----- ΣE_1 | ----- ΣE_2 |
| $\Gamma \vdash (\pi_1, \pi_2) : \Sigma x:A. B$ | | $\Gamma \vdash \mathbf{fst} \pi : A$ | $\Gamma \vdash \mathbf{snd} \pi : [\mathbf{fst} \pi/x] B$ |

Dependent type theory in full generality is very powerful: it is able to express almost any conceivable property of programs directly in the types of the program. This generality comes at a steep price — checking that a given program is of a given type is undecidable. For this reason, dependent type theories in practice do not allow quantification over arbitrary programs, but rather restrict to programs of a given decidable *index domain*, for example integers, strings, or linear programs.

Since dependent type theories allow types to depend on programs, a natural question to ask is whether it is possible for programs to depend on types, or any other combination. There are many kinds of answers to such questions. A popular approach in type theory is to allow programs to be quantified over types, also known as *parametric polymorphism*; of this there are two main kinds: if types and programs are kept separate, then one obtains a somewhat more well-behaved system called *predicative polymorphism*; if the distinction between program and type is blurred, one obtains the type-theoretic analogue of higher-order logic, also known as *impredicative polymorphism*. Various combinations of dependency and polymorphism have been considered in the literature, the most famous being the lambda cube of Henk Barendregt.

The intersection of logic and type theory is a vast and active research area. New logics are usually formalised in a general type theoretic setting, known as a logical framework. Popular modern logical frameworks such as the calculus of constructions and LF are based on higher-order dependent type theory, with various trade-offs in terms of decidability and expressive power. These logical frameworks are themselves always specified as natural deduction systems, which is a testament to the versatility of the natural deduction approach.

Classical and modal logics

For simplicity, the logics presented so far have been intuitionistic. Classical logic extends intuitionistic logic with an additional axiom or principle of excluded middle:

For any proposition p , the proposition $p \vee \neg p$ is true.

This statement is not obviously either an introduction or an elimination; indeed, it involves two distinct connectives. Gentzen's original treatment of excluded middle prescribed one of the following three (equivalent) formulations, which were already present in analogous forms in the systems of Hilbert and Heyting:

| | | |
|-----------------------|-----------------------|------------------------------|
| ----- XM ₁ | $\neg\neg A$ true | |
| $A \vee \neg A$ true | ----- XM ₂ | ----- u |
| | A true | $\neg A$ true |
| | | \square |
| | | p true |
| | | ----- XM ₃ u, p |
| | | A true |

(XM₃ is merely XM₂ expressed in terms of E.) This treatment of excluded middle, in addition to being objectionable from a purist's standpoint, introduces additional complications in the definition of normal forms.

A comparatively more satisfactory treatment of classical natural deduction in terms of introduction and elimination rules alone was first proposed by Parigot in 1992 in the form of a classical lambda calculus called $\lambda\mu$. The key insight of his approach was to replace a truth-centric judgement A true with a more classical notion, reminiscent of the sequent calculus: in localised form, instead of $\Gamma \vdash A$, he used $\Gamma \vdash \Delta$, with Δ a collection of propositions similar to Γ . Γ was treated as a conjunction, and Δ as a disjunction. This structure is essentially lifted directly from classical sequent calculi, but the innovation in $\lambda\mu$ was to give a computational meaning to classical natural deduction proofs in terms of a call/cc or a throw/catch mechanism seen in LISP and its descendants. (See also: first class control.)

Another important extension was for modal and other logics that need more than just the basic judgement of truth. These were first described, for the alethic modal logics S4 and S5, in a natural deduction style by Prawitz in 1965, and have since accumulated a large body of related work. To give a simple example, the modal logic S4 requires one new judgement, " A valid", that is categorical with respect to truth:

If " A true" under no assumptions of the form " B true", then " A valid".

This categorical judgement is internalised as a unary connective $\square A$ (read " $necessarily A$ ") with the following introduction and elimination rules:

| | |
|-------------------|-------------------|
| A valid | $\square A$ true |
| ----- $\square I$ | ----- $\square E$ |
| $\square A$ true | A true |

Note that the premise " A valid" has no defining rules; instead, the categorical definition of validity is used in its place. This mode becomes clearer in the localised form when the hypotheses are explicit. We write " $\Omega; \Gamma \vdash A$ true" where Γ contains the true hypotheses as before, and Ω contains valid hypotheses. On the right there is just a single judgement " A true"; validity is not needed here since " $\Omega \vdash A$ valid" is by definition the same as " $\Omega; \cdot \vdash A$ true". The introduction and elimination forms are then:

| | |
|---|---|
| $\Omega; \cdot \vdash n : A \text{ true}$ | $\Omega; \Gamma \vdash n : \Box A \text{ true}$ |
| ----- $\Box I$ | ----- $\Box E$ |
| $\Omega; \cdot \vdash \mathbf{box} \ n : \Box A \text{ true}$ | $\Omega; \Gamma \vdash \mathbf{unbox} \ n : A \text{ true}$ |

The modal hypotheses have their own version of the hypothesis rule and substitution theorem.

| |
|--|
| ----- valid-hyp |
| $\Omega, u : (A \text{ valid}) ; \Gamma \vdash u : A \text{ true}$ |

Modal substitution theorem

If $\Omega; \cdot \vdash \pi_1 : A \text{ true}$ and $\Omega, u : (A \text{ valid}) ; \Gamma \vdash \pi_2 : C \text{ true}$, then $\Omega; \Gamma \vdash [\pi_1/u] \pi_2 : C \text{ true}$.

This framework of separating judgements into distinct collections of hypotheses, also known as *multi-zoned* or *polyadic* contexts, is very powerful and extensible; it has been applied for many different modal logics, and also for linear and other substructural logics, to give a few examples. However, relatively few systems of modal logic can be formalised directly in natural deduction; to give proof-theoretic characterisations of these systems, extensions such as labelling,^[1] or systems of deep inference^[2]

Comparison with other foundational approaches

Sequent calculus

The sequent calculus is the chief alternative to natural deduction as a foundation of mathematical logic. In natural deduction the flow of information is bi-directional: elimination rules flow information downwards by deconstruction, and introduction rules flow information upwards by assembly. Thus, a natural deduction proof does not have a purely bottom-up or top-down reading, making it unsuitable for automation in proof search, or even for proof checking (or type-checking in type theory). To address this fact, Gentzen in 1935 proposed his sequent calculus, though he initially intended it as a technical device for clarifying the consistency of predicate logic. Kleene, in his seminal 1952 book *Introduction to Metamathematics* (ISBN 0-7204-2103-9), gave the first formulation of the sequent calculus in the modern style.

In the sequent calculus all inference rules have a purely bottom-up reading. Inference rules can apply to elements on both sides of the turnstile. (To differentiate from natural deduction, this article uses a double arrow \Rightarrow instead of the right tack \vdash for sequents.) The introduction rules of natural deduction are viewed as *right rules* in the sequent calculus, and are structurally very similar. The elimination rules on the other hand turn into *left rules* in the sequent calculus. To give an example, consider disjunction; the right rules are familiar:

| | |
|-------------------------------|-------------------------------|
| $\Gamma \Rightarrow A$ | $\Gamma \Rightarrow B$ |
| ----- $\vee R_1$ | ----- $\vee R_2$ |
| $\Gamma \Rightarrow A \vee B$ | $\Gamma \Rightarrow A \vee B$ |

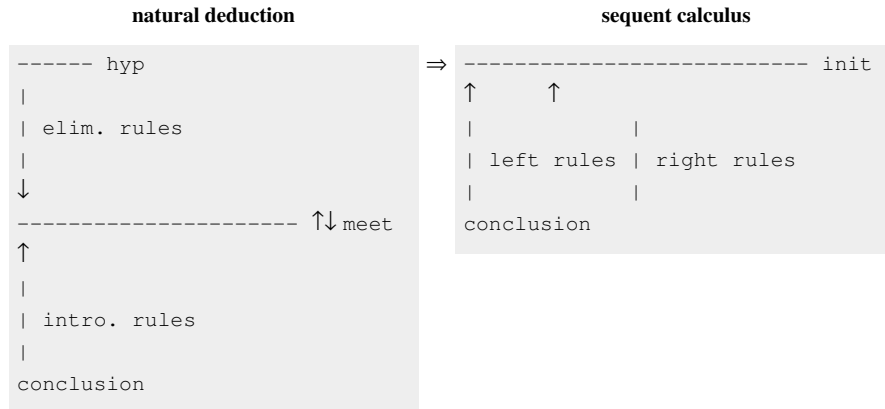
On the left:

| | |
|---------------------------------------|-----------------------------|
| $\Gamma, u:A \Rightarrow C$ | $\Gamma, v:B \Rightarrow C$ |
| ----- $\vee L$ | |
| $\Gamma, w: (A \vee B) \Rightarrow C$ | |

Recall the $\vee E$ rule of natural deduction in localised form:

| | | |
|--------------------------|------------------------|------------------------|
| $\Gamma \vdash A \vee B$ | $\Gamma, u:A \vdash C$ | $\Gamma, v:B \vdash C$ |
| ----- $\vee E$ | | |
| $\Gamma \vdash C$ | | |

The proposition $A \sqcap B$, which is the succedent of a premise in $\vee E$, turns into a hypothesis of the conclusion in the left rule $\vee L$. Thus, left rules can be seen as a sort of inverted elimination rule. This observation can be illustrated as follows:



In the sequent calculus, the left and right rules are performed in lock-step until one reaches the *initial sequent*, which corresponds to the meeting point of elimination and introduction rules in natural deduction. These initial rules are superficially similar to the hypothesis rule of natural deduction, but in the sequent calculus they describe a *transposition* or a *handshake* of a left and a right proposition:

```

----- init
Γ, u:A ⇒ A

```

The correspondence between the sequent calculus and natural deduction is a pair of soundness and completeness theorems, which are both provable by means of an inductive argument.

Soundness of \Rightarrow wrt. \vdash

If $\Gamma \Rightarrow A$, then $\Gamma \vdash A$.

Completeness of \Rightarrow wrt. \vdash

If $\Gamma \vdash A$, then $\Gamma \Rightarrow A$.

It is clear by these theorems that the sequent calculus does not change the notion of truth, because the same collection of propositions remain true. Thus, one can use the same proof objects as before in sequent calculus derivations. As an example, consider the conjunctions. The right rule is virtually identical to the introduction rule

| sequent calculus | | natural deduction |
|--|------------|--|
| <pre> Γ ⇒ π₁ : A Γ ⇒ π₂ : B ----- Γ ⇒ (π₁, π₂) : A ∧ B </pre> | $\wedge R$ | <pre> Γ ⊢ π₁ : A Γ ⊢ π₂ : B ----- Γ ⊢ (π₁, π₂) : A ∧ B </pre> |

The left rule, however, performs some additional substitutions that are not performed in the corresponding elimination rules.

| sequent calculus | natural deduction |
|--|--|
| $\frac{\Gamma, v: (A \wedge B), u:A \Rightarrow \pi : C}{\Gamma, v: (A \wedge B) \Rightarrow [\mathbf{fst} \ v/u] \ \pi : C} \wedge L_1$ | $\frac{\Gamma \vdash \pi : A \wedge B}{\Gamma \vdash \mathbf{fst} \ \pi : A} \wedge E_1$ |
| $\frac{\Gamma, v: (A \wedge B), u:B \Rightarrow \pi : C}{\Gamma, v: (A \wedge B) \Rightarrow [\mathbf{snd} \ v/u] \ \pi : C} \wedge L_2$ | $\frac{\Gamma \vdash \pi : A \wedge B}{\Gamma \vdash \mathbf{snd} \ \pi : B} \wedge E_2$ |

The kinds of proofs generated in the sequent calculus are therefore rather different from those of natural deduction. The sequent calculus produces proofs in what is known as the β -normal η -long form, which corresponds to a canonical representation of the normal form of the natural deduction proof. If one attempts to describe these proofs using natural deduction itself, one obtains what is called the *intercalation calculus* (first described by John Byrnes [3]), which can be used to formally define the notion of a *normal form* for natural deduction.

The substitution theorem of natural deduction takes the form of a structural rule or structural theorem known as *cut* in the sequent calculus.

Cut (substitution)

If $\Gamma \Rightarrow \pi_1 : A$ and $\Gamma, u:A \Rightarrow \pi_2 : C$, then $\Gamma \Rightarrow [\pi_1/u] \pi_2 : C$.

In most well behaved logics, cut is unnecessary as an inference rule, though it remains provable as a meta-theorem; the superfluosness of the cut rule is usually presented as a computational process, known as *cut elimination*. This has an interesting application for natural deduction; usually it is extremely tedious to prove certain properties directly in natural deduction because of an unbounded number of cases. For example, consider showing that a given proposition is *not* provable in natural deduction. A simple inductive argument fails because of rules like $\vee E$ or E which can introduce arbitrary propositions. However, we know that the sequent calculus is complete with respect to natural deduction, so it is enough to show this unprovability in the sequent calculus. Now, if cut is not available as an inference rule, then all sequent rules either introduce a connective on the right or the left, so the depth of a sequent derivation is fully bounded by the connectives in the final conclusion. Thus, showing unprovability is much easier, because there are only a finite number of cases to consider, and each case is composed entirely of sub-propositions of the conclusion. A simple instance of this is the *global consistency* theorem: " $\cdot \vdash \perp$ true" is not provable. In the sequent calculus version, this is manifestly true because there is no rule that can have " $\cdot \Rightarrow \perp$ " as a conclusion! Proof theorists often prefer to work on cut-free sequent calculus formulations because of such properties.

References

Historical references

- Stanislaw Jaśkowski, 1934. *On the Rules of Suppositions in Formal Logic*.
- Gerhard Gentzen, 1934/5. *Untersuchungen uber das logische Schließen* (English translation *Investigations into Logical Deduction* in Szabo)

Textbooks, surveys and co

- Jon Barwise and John Etchemendy, 2000. *Language Proof and Logic*. CSLI (University of Chicago Press) and New York: Seven Bridges Press. A gentle introduction to first-order logic via natural deduction, by two first rate logicians.
- Jean Gallier's excellent tutorial on Constructive Logic and Typed Lambda-Calculi, <ftp://ftp.cis.upenn.edu/pub/papers/gallier/conslog1.ps>.
- Jean-Yves Girard (1990). *Proofs and Types* ^[3]. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, England. Translated and with appendices by Paul Taylor and Yves Lafont.

- Per Martin-Löf (1996). "On the meanings of the logical constants and the justifications of the logical laws" ^[4]. *Nordic Journal of Philosophical Logic* **1** (1): 11–60. Lecture notes to a short course at Università degli Studi di Siena, April 1983.

Other references

- Frank Pfenning and Rowan Davies (2001). "A judgmental reconstruction of modal logic" ^[5]. *Mathematical Structures in Computer Science* **11** (4): 511–540. doi:10.1017/S0960129501003322.
- Alex Simpson, 1993. *The Proof Theory and Semantics of Intuitionistic Modal Logic*. PhD thesis, University of Edinburgh.
- Phiniki Stouppa, 2004. The Design of Modal Proof Theories: The Case of S5. MSc thesis, University of Dresden.

External links

- Clemente, Daniel, "Introduction to natural deduction." ^[6]
- Domino On Acid. ^[7] Natural deduction visualized as a game of dominoes.
- Pelletier, Jeff, "A History of Natural Deduction and Elementary Logic Textbooks." ^[8]

References

- [1] The addition of labels to formulae permits much finer control of the conditions under which rules apply, allowing the more flexible techniques of analytic tableaux to be applied, as has been done in the case of labelled deduction. Labels also allow the naming of worlds in Kripke semantics; Simpson (1993) presents an influential technique for converting frame conditions of modal logics in Kripke semantics into inference rules in a natural deduction formalisation of hybrid logic.
- [2] Stouppa (2004) surveys the application of many proof theories, such as Avron and Pottinger's hypersequents and Belnap's display logic to such modal logics as S5 and B.
- [3] <http://www.cs.man.ac.uk/~pt/stable/Proofs+Types.html>
- [4] <http://docenti.lett.unisi.it/files/4/1/1/6/martinlof4.pdf>
- [5] <http://www-2.cs.cmu.edu/~fp/papers/mcs00.pdf>
- [6] <http://www.danielclemente.com/logica/dn.en.pdf>
- [7] <http://www.winterdrache.de/freeware/domino/>
- [8] <http://www.sfu.ca/~jeffpell/papers/pelletierNDtexts.pdf>

Isabelle (theorem prover)

| | |
|-------------------------|---|
| Stable release | Isabelle 2009-2 |
| Written in | Standard ML |
| Operating system | Linux, Solaris, Mac OS X, Windows (Cygwin) |
| Type | Mathematics |
| License | BSD license |
| Website | http://isabelle.in.tum.de/ |

The **Isabelle theorem prover** is an interactive theorem proving framework, a successor of the Higher Order Logic (HOL) theorem prover. It is an LCF-style theorem prover (written in Standard ML), so it is based on a small logical core guaranteeing logical correctness. Isabelle is generic: it provides a meta-logic (a weak type theory), which is used to encode object logics like First-order logic (FOL), Higher-order logic (HOL) or Zermelo–Fraenkel set theory (ZFC). Isabelle's main proof method is a higher-order version of resolution, based on higher-order unification. Though interactive, Isabelle also features efficient automatic reasoning tools, such as a term rewriting engine and a tableaux prover, as well as various decision procedures. Isabelle has been used to formalize numerous theorems from mathematics and computer science, like Gödel's completeness theorem, Gödel's theorem about the consistency of the axiom of choice, the prime number theorem, correctness of security protocols, and properties of programming language semantics. The Isabelle theorem prover is free software, released under the revised BSD license.

Example proof

Isabelle's proof language Isar aims to support proofs that are both human-readable and machine-checkable. For example, the proof that the square root of two is not rational can be written as follows.

```
theorem sqrt2_not_rational:
  "sqrt (real 2) ∉ ℚ"
proof
  assume "sqrt (real 2) ∈ ℚ"
  then obtain m n :: nat where
    n_nonzero: "n ≠ 0" and sqrt_rat: "|sqrt (real 2)| = real m / real n"
    and lowest_terms: "gcd m n = 1" ..
  from n_nonzero and sqrt_rat have "real m = |sqrt (real 2)| * real n" by simp
  then have "real (m²) = (sqrt (real 2))² * real (n²)" by (auto simp add: power2_eq_square)
  also have "(sqrt (real 2))² = real 2" by simp
  also have "... * real (n²) = real (2 * n²)" by simp
  finally have eq: "m² = 2 * n²" ..
  hence "2 dvd m²" ..
  with two_is_prime have dvd_m: "2 dvd m" by (rule prime_dvd_power_two)
  then obtain k where "m = 2 * k" ..
  with eq have "2 * n² = 2² * k²" by (auto simp add: power2_eq_square mult_ac)
  hence "n² = 2 * k²" by simp
  hence "2 dvd n²" ..
  with two_is_prime have "2 dvd n" by (rule prime_dvd_power_two)
  with dvd_m have "2 dvd gcd m n" by (rule gcd_greatest)
  with lowest_terms have "2 dvd 1" by simp
```

```
thus False by arith
qed
```

Usage

Among other places, Isabelle has been applied by Hewlett-Packard in the design of the HP 9000 line of server's Runway bus where it discovered a number of bugs uncaught by previous testing and simulation.^[1]

More recently, Isabelle has also been used successfully in software implementation verification. In 2009, the L4.verified project at NICTA produced the first formal proof of functional correctness of a general-purpose operating system kernel^[2] : the seL4 (secure embedded L4) microkernel. The proof is constructed and checked in Isabelle/HOL and comprises over 200,000 lines of proof script. The verification covers code, design, and implementation and the main theorem states that the C code correctly implements the formal specification of the kernel. The proof uncovered 160 bugs in the C code of the seL4 kernel, and about 150 issues in each of design and specification.

Larry Paulson keeps a list of research projects^[3] that use Isabelle.

See also

- Formal methods
- Intelligible semi-automated reasoning
- Lightweight Java

References

- Lawrence C. Paulson: *The foundation of a generic theorem prover*. Journal of Automated Reasoning, Volume 5 , Issue 3 (September 1989), Pages: 363-397, ISSN 0168-7433
- Lawrence C. Paulson: *The Isabelle Reference Manual*
- M. A. Ozols, K. A. Eastaughffe, and A. Cant. "DOVE: Design Oriented Verification and Evaluation". *Proceedings of AMAST 97*, M. Johnson, editor, Sydney, Australia. Lecture Notes in Computer Science (LNCS) Vol. 1349, Springer Verlag, 1997.
- Tobias Nipkow, Lawrence C. Paulson, Markus Wenzel: *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*

External links

- Isabelle website^[4]
- The Archive of Formal Proofs^[5]
- IsarMathLib^[6]

References

- [1] Philip Wadler's "An Angry Half-Dozen" (<http://portal.acm.org/citation.cfm?id=274933>) (1998) attributes this result to: Albert J. Camilleri. "A hybrid approach to verifying liveness in a symmetric multiprocessor". 10th International Conference on Theorem Proving in Higher-Order Logics, Elsa Gunter and Amy Felty, editors, Murray Hill, New Jersey, August 1997. Lecture Notes in Computer Science (LNCS) Vol. 1275, Springer Verlag, 1997
- [2] Klein, Gerwin; Elphinstone, Kevin; Heiser, Gernot; Andronick, June; Cock, David; Derrin, Philip; Elkaduwe, Dhammika; Engelhardt, Kai; Kolanski, Rafal; Norrish, Michael; Sewell, Thomas; Tuch, Harvey; Winwood, Simon (October 2009). "seL4: Formal verification of an OS kernel" (<http://www.sigops.org/sosp/sosp09/papers/klein-sosp09.pdf>). . Big Sky, MT, USA. pp. 207-200. .
- [3] <http://www.cl.cam.ac.uk/research/hvg/Isabelle/projects.html>
- [4] <http://isabelle.in.tum.de/>
- [5] <http://afp.sourceforge.net/>
- [6] <http://savannah.nongnu.org/projects/isarmathlib>

Satisfiability Modulo Theories

In computer science, the **Satisfiability Modulo Theories (SMT) problem** is a decision problem for logical formulas with respect to combinations of background theories expressed in classical first-order logic with equality. Examples of theories typically used in computer science are the theory of real numbers, the theory of integers, and the theories of various data structures such as lists, arrays, bit vectors and so on.

Basic terminology

Formally speaking, an SMT instance is a formula in first-order logic, where some function and predicate symbols have additional interpretations, and SMT is the problem of determining whether such a formula is satisfiable. In other words, imagine an instance of the Boolean satisfiability problem (SAT) in which some of the binary variables are replaced by predicates over a suitable set of non-binary variables. A predicate is basically a binary-valued function of non-binary variables. Example predicates include linear inequalities (e.g., $3x + 2y - z \geq 4$) or equalities involving so-called uninterpreted terms and function symbols (e.g., $f(f(u, v), v) = f(u, v)$ where f is some unspecified function of two unspecified arguments.) These predicates are classified according to the theory they belong to. For instance, linear inequalities over real variables are evaluated using the rules of the theory of linear real arithmetic, whereas predicates involving uninterpreted terms and function symbols are evaluated using the rules of the theory of uninterpreted functions with equality (sometimes referred to as the empty theory [1]). Other theories include the theories of arrays and list structures (useful for modeling and verifying software programs), and the theory of bit vectors (useful in modeling and verifying hardware designs). Subtheories are also possible: for example, difference logic is a sub-theory of linear arithmetic in which each inequality is restricted to have the form $x - y \leq c$ for variables x and y and constant c .

Most SMT solvers support only quantifier free fragments of their logics.

Expressive power of SMT

An SMT instance is a generalization of a Boolean SAT instance in which various sets of variables are replaced by predicates from a variety of underlying theories. Obviously, SMT formulas provide a much richer modeling language than is possible with Boolean SAT formulas. For example, an SMT formula allows us to model the datapath operations of a microprocessor at the word rather than the bit level.

SMT solvers

Early attempts for solving SMT instances involved translating them to Boolean SAT instances (e.g., a 32-bit integer variable would be encoded by 32 bit variables with appropriate weights and word-level operations such as 'plus' would be replaced by lower-level logic operations on the bits) and passing this formula to a Boolean SAT solver. This approach, which is referred to as *the eager approach*, has its merits: by pre-processing the SMT formula into an equivalent Boolean SAT formula we can use existing Boolean SAT solvers "as-is" and leverage their performance and capacity improvements over time. On the other hand, the loss of the high-level semantics of the underlying theories means that the Boolean SAT solver has to work a lot harder than necessary to discover "obvious" facts (such as $x + y = y + x$ for integer addition.) This observation led to the development of a number of SMT solvers that tightly integrate the Boolean reasoning of a DPLL-style search with theory-specific solvers (*T-solvers*) that handle conjunctions (ANDs) of predicates from a given theory. This approach is referred to as *the lazy approach*.

Dubbed DPLL(T) (Nieuwenhuis, Oliveras & Tinelli 2006), this architecture gives the responsibility of Boolean reasoning to the DPLL-based SAT solver which, in turn, interacts with a solver for theory T through a well-defined interface. The theory solver need only worry about checking the feasibility of conjunctions of theory predicates passed on to it from the SAT solver as it explores the Boolean search space of the formula. For this integration to

work well, however, the theory solver must be able to participate in propagation and conflict analysis, i.e., it must be able to infer new facts from already established facts, as well as to supply succinct explanations of infeasibility when theory conflicts arise. In other words, the theory solver must be incremental and backtrackable.

CVC (for Cooperating Validity Checker) is an automatic theorem prover for the Satisfiability Modulo Theories problem. It was developed originally by the Stanford Verification Group^[2] as a successor to the Stanford Validity Checker^[3], but is now maintained primarily by researchers at New York University and the University of Iowa. Its latest release is version 3.0 or CVC3. Previous releases were called CVC Lite (v2.0) and CVC (v1.0).

SMT for undecidable theories

Most of the common SMT approaches support decidable theories. However, many real-world systems can only be modelled by means of non-linear arithmetic over the real numbers involving transcendental functions, e.g. an aircraft and its behavior. This fact motivates an extension of the SMT problem to non-linear theories, e.g. determine whether

$$(\sin(x)^3 = \cos(\log(y) \cdot x) \vee b \vee -x^2 \geq 2.3y) \\ \wedge \left(\neg b \vee y < -34.4 \vee \exp(x) > \frac{y}{x} \right)$$

where

$$b \in \mathbb{B}, x, y \in \mathbb{R}$$

is satisfiable. Then, such problems become undecidable in general. (It is important to note, however, that the theory of real closed fields, and thus the full first order theory of the real numbers, are decidable using quantifier elimination. This is due to Alfred Tarski.) The first order theory of the natural numbers with addition (but not multiplication), called Presburger arithmetic, is also decidable. Since multiplication by constants can be implemented as nested additions, the arithmetic in many computer programs can be expressed using Presburger arithmetic, resulting in decidable formulas.

Examples of SMT solvers addressing Boolean combinations of theory atoms from undecidable arithmetic theories over the reals are ABSolver (Bauer, Pister & Tautschnig 2007), which employs a classical DPLL(T) architecture with a non-linear optimization packet as (necessarily incomplete) subordinate theory solver, and HySAT-2, building on a unification of DPLL SAT-solving and interval constraint propagation called the iSAT algorithm (Fränzle et al. 2007).

External links

- SMT-LIB: The Satisfiability Modulo Theories Library^[4]
- SMT-COMP: The Satisfiability Modulo Theories Competition^[5]
- Decision procedures - an algorithmic point of view^[6]

SMT solvers

- ABSolver^[7]
- Barcelogic^[8]
- Beaver^[9]
- Boolector^[10]
- CVC3^[11]
- The Decision Procedure Toolkit (DPT)^[12]
- Alt-Ergo^[13]
- HySAT^[14]
- MathSAT^[15]
- OpenSMT, an Open-source lazy SMT Solver^[16]

- SatEEn^[17]
- Spear^[18]
- STP^[19]
- SWORD^[20]
- veriT^[21]
- Yices^[22]
- Z3^[23]

References

1. Bauer, A.; Pister, M.; Tautschnig, M. (2007), "Tool-support for the analysis of hybrid systems and models", *Proceedings of the 2007 Conference on Design, Automation and Test in Europe (DATE'07)*, IEEE Computer Society, pp. 1, doi:10.1109/DATE.2007.364411
2. Vijay Ganesh, Decision Procedures for Bit-Vectors^[24], Arrays and Integers, Computer Science Department, Stanford University, Stanford, CA, USA, Sept 2007
3. Susmit Jha, Rhishikesh Limaye, and Sanjit A. Seshia. Beaver: Engineering an efficient SMT solver for bit-vector arithmetic.^[25] In *Proceedings of 21st International Conference on Computer-Aided Verification*, pages 668-674, 2009.
4. R. E. Bryant, S. M. German, and M. N. Velev, "Microprocessor Verification Using Efficient Decision Procedures for a Logic of Equality with Uninterpreted Functions," in *Analytic Tableaux and Related Methods*, pp. 1–13, 1999.
5. M. Davis and H. Putnam, A Computing Procedure for Quantification Theory (doi:10.1145/321033.321034), *Journal of the Association for Computing Machinery*, vol. 7, no., pp. 201–215, 1960.
6. M. Davis, G. Logemann, and D. Loveland, A Machine Program for Theorem-Proving (doi:10.1145/368273.368557), *Communications of the ACM*, vol. 5, no. 7, pp. 394–397, 1962.
7. Fränzle, M.; Herde, C.; Ratschan, S.; Schubert, T.; Teige, T. (2007), "Efficient Solving of Large Non-linear Arithmetic Constraint Systems with Complex Boolean Structure"^[26], *JSAT Special Issue on SAT/CP Integration*, **1**, pp. 209–236
8. D. Kroening and O. Strichman, *Decision Procedures — an algorithmic point of view* (2008), Springer (Theoretical Computer Science series) ISBN 978-3540741046.
9. G.-J. Nam, K. A. Sakallah, and R. Rutenbar, A New FPGA Detailed Routing Approach via Search-Based Boolean Satisfiability (doi:10.1109/TCAD.2002.1004311), *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 6, pp. 674–684, 2002.
10. Nieuwenhuis, R.; Oliveras, A.; Tinelli, C. (2006), "Solving SAT and SAT Modulo Theories: From an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T)"^[27], *Journal of the ACM*, **53(6)**, pp. 937–977.


This article is adapted from a column in the ACM SIGDA^[28] e-newsletter^[29] by Prof. Karem Sakallah^[30]
 Original text is available here^[31].

References

- [1] <http://www.csl.sri.com/users/demoura/smt-comp/introduction.shtml>
- [2] <http://verify.stanford.edu>
- [3] <http://verify.stanford.edu/SVC/>
- [4] <http://combination.cs.uiowa.edu/smtlib/>
- [5] <http://www.smtcomp.org>
- [6] <http://www.decision-procedures.org>
- [7] <http://absolver.sourceforge.net/>
- [8] <http://www.lsi.upc.edu/~oliveras/bclt-main.html>
- [9] <http://www.eecs.berkeley.edu/~jha/beaver.html>
- [10] <http://fmv.jku.at/boolector/index.html>

- [11] <http://www.cs.nyu.edu/acsys/cvc3/>
 - [12] <http://sourceforge.net/projects/dpt>
 - [13] <http://ergo.lri.fr/>
 - [14] <http://hysat.informatik.uni-oldenburg.de/>
 - [15] <http://mathsat4.disi.unitn.it/>
 - [16] <http://verify.inf.unisi.ch/opensmt>
 - [17] <http://vlsi.colorado.edu/~hhkim/sateen/>
 - [18] http://www.cs.ubc.ca/~babic/index_spear.htm
 - [19] <http://sites.google.com/site/stpfastprover/>
 - [20] <http://www.informatik.uni-bremen.de/agra/eng/sword.php>
 - [21] <http://www.verit-solver.org/>
 - [22] <http://yices.csl.sri.com/>
 - [23] <http://research.microsoft.com/projects/z3/>
 - [24] http://people.csail.mit.edu/vganesh/Publications_files/vg2007-PhD-STANFORD.pdf
 - [25] http://dx.doi.org/10.1007/978-3-642-02658-4_53
 - [26] http://jsat.ewi.tudelft.nl/content/volume1/JSAT1_11_Fraenzle.pdf
 - [27] <ftp://ftp.cs.uiowa.edu/pub/tinelli/papers/NieOT-JACM-06.pdf>
 - [28] <http://www.sigda.org>
 - [29] <http://www.sigda.org/newsletter/index.html>
 - [30] <http://www.eecs.umich.edu/~karem>
 - [31] http://www.sigda.org/newsletter/2006/eNews_061215.html
-

Prolog

| | |
|---|--|
| Paradigm | Logic programming |
| Appeared in | 1972 |
| Designed by | Alain Colmerauer |
| Major implementations | BProlog, Ciao Prolog, ECLiPSe, GNU Prolog, Logic Programming Associates, Poplog Prolog, P#, Quintus, SICStus, Strawberry, SWI-Prolog, tuProlog, YAP-Prolog |
| Dialects | ISO Prolog, Edinburgh Prolog |
| Influenced | Visual Prolog, Mercury, Oz, Erlang, Strand, KL0, KL1, Datalog |
|  Prolog at Wikibooks | |

Prolog is a general purpose logic programming language associated with artificial intelligence and computational linguistics.

Prolog has its roots in formal logic, and unlike many other programming languages, Prolog is declarative: The program logic is expressed in terms of relations, represented as facts and rules. A computation is initiated by running a *query* over these relations.

The language was first conceived by a group around Alain Colmerauer in Marseille, France, in the early 1970s and the first Prolog system was developed in 1972 by Colmerauer with Phillipe Roussel.^[1]

Prolog was one of the first logic programming languages, and remains among the most popular such languages today, with many free and commercial implementations available. While initially aimed at natural language processing, the language has since then stretched far into other areas like theorem proving, expert systems, games, automated answering systems, ontologies and sophisticated control systems. Modern Prolog environments support the creation of graphical user interfaces, as well as administrative and networked applications.

Syntax and semantics

In Prolog, program logic is expressed in terms of relations, and a computation is initiated by running a *query* over these relations. Relations and queries are constructed using Prolog's single data type, the *term*. Relations are defined by *clauses*. Given a query, the Prolog engine attempts to find a resolution refutation of the negated query. If the negated query can be refuted, i.e., an instantiation for all free variables is found that makes the union of clauses and the singleton set consisting of the negated query false, it follows that the original query, with the found instantiation applied, is a logical consequence of the program. This makes Prolog (and other logic programming languages) particularly useful for database, symbolic mathematics, and language parsing applications. Because Prolog allows impure predicates, checking the truth value of certain special predicates may have some deliberate side effect, such as printing a value to the screen. Because of this the programmer is permitted to use some amount of conventional imperative programming when the logical paradigm is inconvenient. It has a purely logical subset, called "pure Prolog", as well as a number of extralogical features.

Data types

Prolog's single data type is the *term*. Terms are either *atoms*, *numbers*, *variables* or *compound terms*.

- An **atom** is a general-purpose name with no inherent meaning. Examples of atoms include `x`, `blue`, `'Taco'`, and `'some atom'`.
- **Numbers** can be floats or integers.
- **Variables** are denoted by a string consisting of letters, numbers and underscore characters, and beginning with an upper-case letter or underscore. Variables closely resemble variables in logic in that they are placeholders for arbitrary terms.
- A **compound term** is composed of an atom called a "functor" and a number of "arguments", which are again terms. Compound terms are ordinarily written as a functor followed by a comma-separated list of argument terms, which is contained in parentheses. The number of arguments is called the term's arity. An atom can be regarded as a compound term with arity zero. Examples of compound terms are `truck_year('Mazda', 1986)` and `'Person_Friends'(zelda,[tom,jim])`.

Special cases of compound terms:

- A *List* is an ordered collection of terms. It is denoted by square brackets with the terms separated by commas or in the case of the empty list, `[]`. For example `[1,2,3]` or `[red,green,blue]`.
- *Strings*: A sequence of characters surrounded by quotes is equivalent to a list of (numeric) character codes, generally in the local character encoding or Unicode if the system supports Unicode. For example, `"to be, or not to be"`.

Rules and facts

Prolog programs describe relations, defined by means of clauses. Pure Prolog is restricted to Horn clauses. There are two types of clauses: Facts and rules. A rule is of the form

```
Head :- Body.
```

and is read as "Head is true if Body is true". A rule's body consists of calls to predicates, which are called the rule's **goals**. The built-in predicate `,/2` (meaning a 2-arity operator with name `,`) denotes conjunction of goals, and `;/2` denotes disjunction. Conjunctions and disjunctions can only appear in the body, not in the head of a rule.

Clauses with empty bodies are called **facts**. An example of a fact is:

```
cat(tom).
```

which is equivalent to the rule:

```
cat(tom) :- true.
```

The built-in predicate `true/0` is always true.

Given the above fact, one can ask:

is tom a cat?

```
?- cat(tom).
Yes
```

what things are cats?

```
?- cat(X).
X = tom
```

Due to the relational nature of many built-in predicates, they can typically be used in several directions. For example, `length/2` can be used to determine the length of a list (`length(List, L)`, given a list `List`) as well as to generate a list skeleton of a given length (`length(X, 5)`), and also to generate both list skeletons and their lengths together (`length(X, L)`). Similarly, `append/3` can be used both to append two lists (`append(ListA, ListB, X)` given lists `ListA` and `ListB`) as well as to split a given list into parts (`append(X, Y, List)`, given a list `List`). For this reason, a comparatively small set of library predicates suffices for many Prolog programs.

As a general purpose language, Prolog also provides various built-in predicates to perform routine activities like input/output, using graphics and otherwise communicating with the operating system. These predicates are not given a relational meaning and are only useful for the side-effects they exhibit on the system. For example, the predicate `write/1` displays a term on the screen.

Evaluation

Execution of a Prolog program is initiated by the user's posting of a single goal, called the query. Logically, the Prolog engine tries to find a resolution refutation of the negated query. The resolution method used by Prolog is called SLD resolution. If the negated query can be refuted, it follows that the query, with the appropriate variable bindings in place, is a logical consequence of the program. In that case, all generated variable bindings are reported to the user, and the query is said to have succeeded. Operationally, Prolog's execution strategy can be thought of as a generalization of function calls in other languages, one difference being that multiple clause heads can match a given call. In that case, the system creates a choice-point, unifies the goal with the clause head of the first alternative, and continues with the goals of that first alternative. If any goal fails in the course of executing the program, all variable bindings that were made since the most recent choice-point was created are undone, and execution continues with the next alternative of that choice-point. This execution strategy is called chronological backtracking. For example:

```
mother_child(trude, sally).

father_child(tom, sally).
father_child(tom, erica).
father_child(mike, tom).

sibling(X, Y)      :- parent_child(Z, X), parent_child(Z, Y).

parent_child(X, Y) :- father_child(X, Y).
parent_child(X, Y) :- mother_child(X, Y).
```

This results in the following query being evaluated as true:

```
?- sibling(sally, erica).
Yes
```

This is obtained as follows: Initially, the only matching clause-head for the query `sibling(sally, erica)` is the first one, so proving the query is equivalent to proving the body of that clause with the appropriate variable bindings in place, i.e., the conjunction (`parent_child(Z,sally), parent_child(Z,erica)`). The next goal to be proved is the leftmost one of this conjunction, i.e., `parent_child(Z, sally)`. Two clause heads match this goal. The system creates a choice-point and tries the first alternative, whose body is `father_child(Z, sally)`. This goal can be proved using the fact `father_child(tom, sally)`, so the binding `Z = tom` is generated, and the next goal to be proved is the second part of the above conjunction: `parent_child(tom, erica)`. Again, this can be proved by the corresponding fact. Since all goals could be proved, the query succeeds. Since the query contained no variables, no bindings are reported to the user. A query with variables, like:

```
?- father_child(Father, Child).
```

enumerates all valid answers on backtracking.

Notice that with the code as stated above, the query `?- sibling(sally, sally).` also succeeds. One would insert additional goals to describe the relevant restrictions, if desired.

Loops and recursion

Iterative algorithms can be implemented by means of recursive predicates.

Negation

The built-in Prolog predicate `\+/1` provides negation as failure, which allows for non-monotonic reasoning. The goal `\+ illegal(X)` in the rule

```
legal(X) :- \+ illegal(X).
```

is evaluated as follows: Prolog attempts to prove the `illegal(X)`. If a proof for that goal can be found, the original goal (i.e., `\+ illegal(X)`) fails. If no proof can be found, the original goal succeeds. Therefore, the `\+/1` prefix operator is called the "not provable" operator, since the query `?- \+ Goal.` succeeds if `Goal` is not provable. This kind of negation is sound if its argument is "ground" (i.e. contains no variables). Soundness is lost if the argument contains variables and the proof procedure is complete. In particular, the query `?- legal(X).` can now not be used to enumerate all things that are legal.

Examples

Here follow some example programs written in Prolog.

Hello world

An example of a query:

```
?- write('Hello world!'), nl.
Hello world!
true.

?-
```

Compiler optimization

Any computation can be expressed declaratively as a sequence of state transitions. As an example, an optimizing compiler with three optimization passes could be implemented as a relation between an initial program and its optimized form:

```
program_optimized(Prog0, Prog) :-
    optimization_pass_1(Prog0, Prog1),
    optimization_pass_2(Prog1, Prog2),
    optimization_pass_3(Prog2, Prog).
```

or equivalently using DCG notation:

```
program_optimized --> optimization_pass_1, optimization_pass_2,
optimization_pass_3.
```

QuickSort

The QuickSort sorting algorithm, relating a list to its sorted version:

```
partition([], _, [], []).
partition([X|Xs], Pivot, Smalls, Bigs) :-
    (   X @< Pivot ->
        Smalls = [X|Rest],
        partition(Xs, Pivot, Rest, Bigs)
    ;   Bigs = [X|Rest],
        partition(Xs, Pivot, Smalls, Rest)
    ).

quicksort([])      --> [].
quicksort([X|Xs]) -->
    { partition(Xs, X, Smaller, Bigger) },
    quicksort(Smaller), [X], quicksort(Bigger).
```

Dynamic programming

The following Prolog program uses dynamic programming to find the longest common subsequence of two lists in polynomial time. The clause database is used for memoization:

```
:- dynamic(stored/1).

memo(Goal) :- ( stored(Goal) -> true ; Goal, assertz(stored(Goal)) ).

lcs([], _, []) :- !.
lcs(_, [], []) :- !.
lcs([X|Xs], [X|Ys], [X|Ls]) :- !, memo(lcs(Xs, Ys, Ls)).
lcs([X|Xs], [Y|Ys], Ls) :-
    memo(lcs([X|Xs], Ys, Ls1)), memo(lcs(Xs, [Y|Ys], Ls2)),
    length(Ls1, L1), length(Ls2, L2),
    (   L1 >= L2 -> Ls = Ls1 ; Ls = Ls2 ).
```

Example query:

```
?- lcs([x,m,j,y,a,u,z], [m,z,j,a,w,x,u], Ls).
Ls = [m, j, a, u]
```

Modules

For programming in the large, Prolog provides a module system. The module system is standardised by ISO.^[2] However, not all Prolog compilers support modules and there are compatibility problems between the module systems of the major Prolog compilers.^[3] Consequently, modules written on one Prolog compiler will not necessarily work on others.

Parsing

There is a special notation called definite clause grammars (DCGs). A rule defined via `-->/2` instead of `:-/2` is expanded by the preprocessor (`expand_term/2`, a facility analogous to macros in other languages) according to a few straightforward rewriting rules, resulting in ordinary Prolog clauses. Most notably, the rewriting equips the predicate with two additional arguments, which can be used to implicitly thread state around, analogous to monads in other languages. DCGs are often used to write parsers or list generators, as they also provide a convenient interface to list differences.

Parser example

A larger example will show the potential of using Prolog in parsing.

Given the sentence expressed in Backus-Naur Form:

```
<sentence>      ::= <stat_part>
<stat_part>     ::= <statement> | <stat_part> <statement>
<statement>     ::= <id> = <expression> ;
<expression>    ::= <operand> | <expression> <operator> <operand>
<operand>       ::= <id> | <digit>
<id>            ::= a | b
<digit>         ::= 0..9
<operator>      ::= + | - | *
```

This can be written in Prolog using DCGs, corresponding to a predictive parser with one token look-ahead:

```
sentence(S)          --> statement(S0), sentence_r(S0, S).
sentence_r(S, S)     --> [].
sentence_r(S0, seq(S0, S)) --> statement(S1), sentence_r(S1, S).

statement(assign(Id,E)) --> id(Id), [=], expression(E), [;].

expression(E) --> term(T), expression_r(T, E).
expression_r(E, E) --> [].
expression_r(E0, E) --> [+], term(T), expression_r(plus(E0,T), E).
expression_r(E0, E) --> [-], term(T), expression_r(minus(E0, T), E).

term(T)             --> factor(F), term_r(F, T).
term_r(T, T)        --> [].
term_r(T0, T)       --> [*], factor(F), term_r(times(T0, F), T).

factor(id(ID))      --> id(ID).
factor(digit(D))    --> [D], { (number(D) ; var(D)), between(0, 9, D) }.
```

```
id(a) --> [a].
id(b) --> [b].
```

This code defines a relation between a sentence (given as a list of tokens) and its abstract syntax tree (AST). Example query:

```
?- phrase(sentence(AST), [a,=,1,+,3,*,b,;,b,=,0,;]).
AST = seq(assign(a, plus(digit(1), times(digit(3), id(b)))), assign(b, digit(0))) ;
```

The AST is represented using Prolog terms and can be used to apply optimizations, to compile such expressions to machine-code, or to directly interpret such statements. As is typical for the relational nature of predicates, these definitions can be used both to parse and generate sentences, and also to check whether a given tree corresponds to a given list of tokens. Using iterative deepening for fair enumeration, each arbitrary but fixed sentence and its corresponding AST will be generated eventually:

```
?- length(Tokens, _), phrase(sentence(AST), Tokens).
Tokens = [a, =, a, (;)], AST = assign(a, id(a)) ;
Tokens = [a, =, b, (;)], AST = assign(a, id(b))
etc.
```

Higher-order programming

First-order logic does not allow quantification over predicates. A higher-order predicate is a predicate that takes one or more other predicates as arguments. Since arbitrary Prolog goals can be constructed and evaluated at run-time, it is easy to write higher-order predicates like `maplist/2`, which applies an arbitrary predicate to each member of a given list, and `sublist/3`, which filters elements that satisfy a given predicate, also allowing for currying.

To convert solutions from temporal representation (answer substitutions on backtracking) to spatial representation (terms), Prolog has various all-solutions predicates that collect all answer substitutions of a given query in a list. This can be used for list comprehension. For example, perfect numbers equal the sum of their proper divisors:

```
perfect(N) :-
    between(1, inf, N), U is N // 2,
    findall(D, (between(1,U,D), N mod D == 0), Ds),
    sumlist(Ds, N).
```

This can be used to enumerate perfect numbers, and also to check whether a number is perfect.

Meta-interpreters and reflection

Prolog is a homoiconic language and provides many facilities for reflection. Its implicit execution strategy makes it possible to write a concise meta-circular evaluator (also called *meta-interpreter*) for pure Prolog code. Since Prolog programs are themselves sequences of Prolog terms (`:-/2` is an infix operator) that are easily read and inspected using built-in mechanisms (like `read/1`), it is easy to write customized interpreters that augment Prolog with domain-specific features.

Turing completeness

Pure Prolog is based on a subset of first-order predicate logic, Horn clauses, which is Turing-complete. Turing completeness of Prolog can be shown by using it to simulate a Turing machine:

```
turing(Tape0, Tape) :-
    perform(q0, [], Ls, Tape0, Rs),
    reverse(Ls, Ls1),
    append(Ls1, Rs, Tape).

perform(qf, Ls, Ls, Rs, Rs) :- !.
perform(Q0, Ls0, Ls, Rs0, Rs) :-
    symbol(Rs0, Sym, RsRest),
    once(rule(Q0, Sym, Q1, NewSym, Action)),
    action(Action, Ls0, Ls1, [NewSym|RsRest], Rs1),
    perform(Q1, Ls1, Ls, Rs1, Rs).

symbol([], b, []).
symbol([Sym|Rs], Sym, Rs).

action(left, Ls0, Ls, Rs0, Rs) :- left(Ls0, Ls, Rs0, Rs).
action(stay, Ls, Ls, Rs, Rs).
action(right, Ls0, [Sym|Ls0], [Sym|Rs], Rs).

left([], [], Rs0, [b|Rs0]).
left([L|Ls], Ls, Rs, [L|Rs]).
```

A simple example Turing machine is specified by the facts:

```
rule(q0, 1, q0, 1, right).
rule(q0, b, qf, 1, stay).
```

This machine performs incrementation by one of a number in unary encoding: It loops over any number of "1" cells and appends an additional "1" at the end. Example query and result:

```
?- turing([1,1,1], Ts).
Ts = [1, 1, 1, 1] ;
```

This illustrates how any computation can be expressed declaratively as a sequence of state transitions, implemented in Prolog as a relation between successive states of interest.

Implementation

ISO Prolog

The ISO Prolog standard consists of two parts. ISO/IEC 13211-1^[4], published in 1995, aims to standardize the existing practices of the many implementations of the core elements of Prolog. It has clarified aspects of the language that were previously ambiguous and leads to portable programs. ISO/IEC 13211-2^[4], published in 2000, adds support for modules to the standard. The standard is maintained by the ISO/IEC JTC1/SC22/WG17^[5] working group. ANSI X3J17 is the US Technical Advisory Group for the standard.^[6]

Compilation

For efficiency, Prolog code is typically compiled to abstract machine code, often influenced by the register-based Warren Abstract Machine (WAM) instruction set. Some implementations employ abstract interpretation to derive type and mode information of predicates at compile time, or compile to real machine code for high performance. Devising efficient implementation techniques for Prolog code is a field of active research in the logic programming community, and various other execution techniques are employed in some implementations. These include clause binarization and stack-based virtual machines.

Tail recursion

Prolog systems typically implement a well-known optimization technique called tail call optimization (TCO) for deterministic predicates exhibiting tail recursion or, more generally, tail calls: A clause's stack frame is discarded before performing a call in a tail position. Therefore, deterministic tail-recursive predicates are executed with constant stack space, like loops in other languages.

Tabling

Some Prolog systems, (BProlog, XSB and Yap), implement an extension called *tabling*, which frees the user from manually storing intermediate results.

Implementation in hardware

During the Fifth Generation Computer Systems project, there were attempts to implement Prolog in hardware with the aim of achieving faster execution with dedicated architectures.^{[7] [8] [9]} Furthermore, Prolog has a number of properties that may allow speed-up through parallel execution.^[10] A more recent approach has been to compile restricted Prolog programs to a field programmable gate array.^[11] However, rapid progress in general-purpose hardware has consistently overtaken more specialised architectures.

Criticism

Although Prolog is widely used in research and education, Prolog and other logic programming languages have not had a significant impact on the computer industry in general.^[12] Most applications are small by industrial standards with few exceeding 100,000 lines of code.^{[13] [14]} Programming in the large is considered to be complicated because not all Prolog compilers support modules, and there are compatibility problems between the module systems of the major Prolog compilers.^[15] Portability of Prolog code across implementations has also been a problem but developments since 2007 have meant: "the portability within the family of Edinburgh/Quintus derived Prolog implementations is good enough to allow for maintaining portable real-world applications."^[16]

Software developed in Prolog has been criticised for having a high performance penalty, but advances in implementation have made some of these arguments obsolete.^[17]

Extensions

Various implementations have been developed from Prolog to extend logic programming capabilities in numerous directions. These include constraint logic programming (CLP), object-oriented logic programming (OOLP), concurrency, Linear Logic (LLP), functional and higher-order logic programming capabilities, plus interoperability with knowledge bases:

Constraints

Constraint logic programming is important for many Prolog applications in industrial settings, like time tabling and other scheduling tasks. Most Prolog systems ship with at least one constraint solver for finite domains, and often also with solvers for other domains like rational numbers.

Higher-order programming

HiLog and λ Prolog extend Prolog with higher-order programming features.

Object orientation

Logtalk is an object-oriented logic programming language that can use most Prolog implementations as a back-end compiler. As a multi-paradigm language, it includes support for both prototypes and classes, protocols (interfaces), component-based programming through category-based composition, event-driven programming, and high-level multi-threading programming.

Oblog is a small, portable, Object-oriented extension to Prolog by Margaret McDougall of EdCAAD, University of Edinburgh.

Concurrency

Prolog-MPI is an open-source SWI-Prolog extension for distributed computing over the Message Passing Interface.^[18] Also there are various concurrent Prolog programming languages.^[19]

Web programming

Some Prolog implementations, notably SWI-Prolog, support server-side web programming with support for web protocols, HTML and XML.^[20] There are also extensions to support semantic web formats such as RDF and OWL.^{[21] [22]} Prolog has also been suggested as a client-side language.^[23]

Other

- F-logic extends Prolog with frames/objects for knowledge representation.
- OW Prolog has been created in order to answer Prolog's lack of graphics and interface.
- Cedar^[24] free and basic prolog interpreter.

Interfaces to other languages

Frameworks exist which can provide a bridge between Prolog and the Java programming language:

- JPL is a bi-directional Java Prolog bridge which ships with SWI-Prolog by default, allowing Java and Prolog to call each other respectively. It is known to have good concurrency support and is under active development.
 - InterProlog^[25], a programming library bridge between Java and Prolog, implementing bi-directional predicate/method calling between both languages. Java objects can be mapped into Prolog terms and vice-versa. Allows the development of GUIs and other functionality in Java while leaving logic processing in the Prolog layer. Supports XSB, SWI-Prolog and YAP.
-

- Prova provides native syntax integration with Java, agent messaging and reaction rules. Prova positions itself as a rule-based scripting (RBS) system for middleware. The language breaks new ground in combining imperative and declarative programming.
- PROL^[26] An embeddable Prolog engine for Java. It includes a small IDE and a few libraries.

Related languages

- The Gödel programming language is a strongly-typed implementation of Concurrent constraint logic programming. It is built on SICStus Prolog.
- Visual Prolog, also formerly known as **PDC Prolog** and Turbo Prolog. Visual Prolog is a strongly typed object-oriented dialect of Prolog, which is considerably different from standard Prolog. As Turbo Prolog it was marketed by Borland, but it is now developed and marketed by the Danish firm PDC (Prolog Development Center) that originally produced it.
- Datalog is a subset of Prolog. It is limited to relationships that may be stratified and does not allow compound terms. In contrast to Prolog, Datalog is not Turing-complete.
- CSC GraphTalk is a proprietary implementation of Warren's Abstract Machine, with additional object-oriented properties.
- In some ways Prolog is a subset of Planner. The ideas in Planner were later further developed in the Scientific Community Metaphor.

History

The name *Prolog* was chosen by Philippe Roussel as an abbreviation for *programmation en logique* (French for *programming in logic*). It was created around 1972 by Alain Colmerauer with Philippe Roussel, based on Robert Kowalski's procedural interpretation of Horn clauses. It was motivated in part by the desire to reconcile the use of logic as a declarative knowledge representation language with the procedural representation of knowledge that was popular in North America in the late 1960s and early 1970s. According to Robert Kowalski, the first Prolog system was developed in 1972 by Alain Colmerauer and Phillipe Roussel.^[27] The first implementations of Prolog were interpreters, however, David H. D. Warren created the Warren Abstract Machine, an early and influential Prolog compiler which came to define the "Edinburgh Prolog" dialect which served as the basis for the syntax of most modern implementations.

Much of the modern development of Prolog came from the impetus of the fifth generation computer systems project (FGCS), which developed a variant of Prolog named *Kernel Language* for its first operating system.

Pure Prolog was originally restricted to the use of a resolution theorem prover with Horn clauses of the form:

$$H :- B_1, \dots, B_n.$$

The application of the theorem-prover treats such clauses as procedures:

$$\text{to show/solve } H, \text{ show/solve } B_1 \text{ and } \dots \text{ and } B_n.$$

Pure Prolog was soon extended, however, to include negation as failure, in which negative conditions of the form $\text{not}(B_i)$ are shown by trying and failing to solve the corresponding positive conditions B_i .

Subsequent extensions of Prolog by the original team introduced Constraint logic programming abilities into the implementations.

See also

- Comparison of Prolog implementations

References

- William F. Clocksin, Christopher S. Mellish: *Programming in Prolog: Using the ISO Standard*. Springer, 5th ed., 2003, ISBN 978-3540006787. (*This edition is updated for ISO Prolog. Previous editions described Edinburgh Prolog.*)
- William F. Clocksin: *Clause and Effect. Prolog Programming for the Working Programmer*. Springer, 2003, ISBN 978-3540629719.
- Michael A. Covington, Donald Nute, Andre Vellino, *Prolog Programming in Depth*, 1996, ISBN 0-13-138645-X.
- Michael A. Covington, *Natural Language Processing for Prolog Programmers*, 1994, ISBN 0-13-62921
- Robert Smith, John Gibson, Aaron Sloman: 'POPLOG's two-level virtual machine support for interactive languages', in *Research Directions in Cognitive Science Volume 5: Artificial Intelligence*, Eds D. Sleeman and N. Bernsen, Lawrence Erlbaum Associates, pp 203-231, 1992.
- Leon Sterling and Ehud Shapiro, *The Art of Prolog: Advanced Programming Techniques*, 1994, ISBN 0-262-19338-8.
- Ivan Bratko, *PROLOG Programming for Artificial Intelligence*, 2000, ISBN 0-201-40375-7.
- Robert Kowalski, *The Early Years of Logic Programming* ^[28], CACM January 1988.
- ISO/IEC 13211: *Information technology — Programming languages — Prolog*. International Organization for Standardization, Geneva.
- Alain Colmerauer and Philippe Roussel, *The birth of Prolog* ^[29], in *The second ACM SIGPLAN conference on History of programming languages*, p. 37-52, 1992.
- Richard O'Keefe, *The Craft of Prolog*, ISBN 0-262-15039-5.
- Patrick Blackburn, Johan Bos, Kristina Striegnitz, *Learn Prolog Now!*, 2006, ISBN 1-904987-17-6.
- David H D Warren, Luis M. Pereira and Fernando Pereira, Prolog - the language and its implementation compared with Lisp. ACM SIGART Bulletin archive, Issue 64. Proceedings of the 1977 symposium on Artificial intelligence and programming languages, pp 109 - 115.

[1] Kowalski, R. A.. *The early years of logic programming*.

[2] ISO/IEC 13211-2: Modules.

[3] Paulo Moura, Logtalk in Association of Logic Programming Newsletter. Vol 17 n. 3, August 2004. (<http://www.cs.kuleuven.ac.be/~dtai/projects/ALP/newsletter/aug04/nav/print/all.html#logtalk>)

[4] ISO/IEC 13211: Information technology — Programming languages — Prolog. International Organization for Standardization, Geneva.

[5] WG17 Working Group (<http://www.sju.edu/~jhodgson/wg17/>)

[6] X3J17 Committee (<http://www.sju.edu/~jhodgson/x3j17.html>)

[7] doi:10.1145/30350.30362

[8] doi:10.1007/3-540-16492-8_73

[9] doi:10.1145/36205.36195

[10] doi:10.1145/504083.504085

[11] <http://www.cl.cam.ac.uk/~am21/research/sa/byrdbbox.ps.gz>

[12] Logic programming for the real world. Zoltan Somogyi, Fergus Henderson, Thomas Conway, Richard O'Keefe. Proceedings of the ILPS'95 Postconference Workshop on Visions for the Future of Logic Programming.

[13] *ibid*

[14] The Prolog 1000 database <http://www.faqs.org/faqs/prolog/resource-guide/part1/section-9.html>

[15] Paulo Moura, Logtalk in Association of Logic Programming Newsletter. Vol 17 n. 3, August 2004. (<http://www.cs.kuleuven.ac.be/~dtai/projects/ALP/newsletter/aug04/nav/print/all.html#logtalk>)

[16] Jan Wielemaker and Vitor Santos Costa: Portability of Prolog programs: theory and case-studies (<http://www.swi-prolog.org/download/publications/porting.pdf>). CICLOPS-WLPE Workshop 2010 (<http://www.floc-conference.org/CICLOPS-WLPE-accepted.html>).

[17] Leon Sterling: *The Practice of Prolog*. 1990, page 32.

[18] <http://apps.lumii.lv/prolog-mpi/>

[19] *Ehud Shapiro. **The family of concurrent logic programming languages** ACM Computing Surveys. September 1989.

[20] doi:10.1017/S1471068407003237

- [21] Jan Wielemaker and Michiel Hildebrand and Jacco van Ossenbruggen (2007), S.Heymans, A. Polleres, E. Ruckhaus, D. Pearce, and G. Gupta, ed., "Using {Prolog} as the fundament for applications on the semantic web" (http://ftp.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-287/paper_1.pdf), *Proceedings of the 2nd Workshop on Applications of Logic Programming and to the web, Semantic Web and Semantic Web Services*, CEUR Workshop Proceedings (Porto, Portugal: CEUR-WS.org) **287**: 84--98,
- [22] Processing OWL2 Ontologies using Thea: An Application of Logic Programming (http://ceur-ws.org/Vol-529/owl2009_submission_43.pdf). Vangelis Vassiliadis, Jan Wielemaker and Chris Mungall. Proceedings of the 5th International Workshop on OWL: Experiences and Directions (OWLED 2009), Chantilly, VA, United States, October 23-24, 2009
- [23] doi:10.1017/S1471068401001211
- [24] <http://sites.google.com/site/cedarprolog/>
- [25] <http://www.declarativa.com/interprolog/>
- [26] <http://igormaznitsa.com/projects/prolog/index.html>
- [27] Kowalski, R. A.. *The early years of logic programming*.
- [28] <http://www.doc.ic.ac.uk/~rak/papers/the%20early%20years.pdf>
- [29] <http://www.lim.univ-mrs.fr/~colmer/ArchivesPublications/HistoireProlog/19novembre92.pdf>

External links

- comp.lang.prolog FAQ (<http://www.logic.at/prolog/faq/>)
- Prolog: The ISO standard (<http://pauillac.inria.fr/~deransar/prolog/docs.html>)
- DECsystem-10 Prolog User's Manual (<http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/prolog/doc/intro/prolog.doc>) (plain text) describes a typical Edinburgh Prolog
- Prolog Tutorial (http://www.csupomona.edu/~jrfisher/www/prolog_tutorial/contents.html) by J.R.Fisher
- Runnable examples (<http://www.allisons.org/ll/Logic/Prolog/Examples/>) by Lloyd Allison
- On-line guide to Prolog Programming (<http://kti.ms.mff.cuni.cz/~bartak/prolog/index.html>) by Roman Bartak
- Learn Prolog Now! (<http://www.learnprolognow.org/>) by Patrick Blackburn, Johan Bos and Kristina Striegnitz
- Prolog and Logic Programming (http://www.cs.bham.ac.uk/~pjh/prolog_course/se207.html) by Dr Peter Hancox
- Building Expert Systems in Prolog (<http://www.amzi.com/ExpertSystemsInProlog/index.htm>), online book by Amzi! Inc.
- Literate programming in Prolog (http://en.literateprograms.org/Category:Programming_language:Prolog)
- Object Oriented Language: Prolog, OOLP and other extensions (http://www.cetus-links.org/oo_prolog.html) by Richard Katz

Article Sources and Contributors

Algebraic structure *Source:* <http://en.wikipedia.org/w/index.php?oldid=366014989> *Contributors:* Abcdefghijk99, Adriatikus, Alexey Feldgendler, Algebraist, AnakngAraw, AxelBoldt, Berria, Bgohla, CRGreathouse, Charles Matthews, Charvest, Chromaticity, Crasshopper, Cronholm144, Danny, David Eppstein, Dmharvey, DustinBernard, Dysprosia, Estevoaei, FlyHigh, Franamax, Propuff, G716, Galaxiaad, Giftlite, Goldencako, Grafen, Grubber, Hans Adler, HeikoEvermann, Ht686rg90, I paton, Icep, Icey, Ideyal, IronGargoyle, Jakob.scholbach, Jayden54, Jeepday, Johnfuhrmann, Jorgen W, Josh Parris, Jshadias, Kuratowski's Ghost, Kusma, Lethe, Marvinfreeman, MaxSem, Melchoir, Mets501, Michael Hardy, Michal.burda, Modify, Msh210, Mysdaao, Naddy, Netrapt, Nishantjr, Obradovic Goran, Olaf, Paul August, Puchiko, RedWolf, Revolver, RexNL, Reyk, Salix alba, Simon12, SoroSuub1, Spiderboy, Staecker, Squirrel, Tobias Bergemann, Toby Bartels, Tomaxer, Tompw, Tristanreid, Trovatore, Varuna, Waltpohl, Wshun, Zinoviev, Zundark, بِنَام, 207 anonymous edits

Mathematical logic *Source:* <http://en.wikipedia.org/w/index.php?oldid=370858737> *Contributors:* Aeosynth, Aleph4, Allan McInnes, Am Fiosaigear, Art LaPella, Atomique, Avaya1, AxelBoldt, Barnaby dawson, Beefalo, Bidabadi, Bjankuloski06en, Blah3, Bomac, Brian G. Wilson, Butane Goddess, CBM, CBM2, CRGreathouse, Chalst, Charles Matthews, Chira, Claygate, Compellingelegance, Conversion script, Crempuff222, CuriousOne, Dan Gluck, David Shay, DavidCBryant, Davin, Dcoetzee, December21st2012Freak, Dgies, Dysprosia, Eduardoporcher, EugeneZelenko, Flemnira, FrankTobia, Gandalf61, Giftlite, Gregbard, Guardian of Light, Hairu Dude, Ham and bacon, Hans Adler, HedgeHog, Hotfeba, Icarus3, Iridescent, Irri"ti"nal, IsleLaMotte, Ivan Bajlo, JLL, Jagged 85, Jersey Devil, Jojit fb, Jon Awbrey, Julian Mendez, KDesk, KSchutte, Kntg, LAX, Lambiam, LarRan, Lebob, Leibniz, Lethe, Ling.Nut, Logic2go, Lycurgus, MK8, Malcolm, MathMartin, Matt Crypto, Meeples, Meloman, Metamusing, Michael Hardy, Mindmatrix, Modster, Mrdthree, Msh210, Musteval, Mwoolf, Myasuda, Nabeth, Nicke Lilltroll, Nigosh, Nixeagle, Nohat, Obradovic Goran, Ojigiri, Otto ter Haar, Ozob, Palaeovia, Paul August, Pcap, PergolesiCoffee, Persian Poet Gal, Piolinfax, Polyvix, Popopp, Qiemem, Razimantv, Recentchanges, Reinis, Renamed user 4, Reyk, Rick Norwood, Roberbyrne, Robin S, RockRockOn, Romanm, Rover6891, Rsimmonds01, Ruud Koot, Sa'y, Salix alba, Samohyl Jan, SaxTeacher, ScNewcastle, Sholto Maud, Skolemizer, Sligocki, Smimram, Smithpith, Snem, Spayrard, SpuriousQ, Stevenmitchell, Stotr, TakuyaMurata, Tales23, TeH nOmInAtOr, The Teatrast, The Wiki ghost, The undertow, TheIndianWikiEditor, Tillmo, Tkos, Tkuvho, Tobias Bergemann, Tony1, Tregoweth, Trovatore, Valeria.depaiva, VictorAnyakin, Wavelength, Wclark, WikiSBTR, Zhentmdfan, 167 anonymous edits

Structure (mathematical logic) *Source:* <http://en.wikipedia.org/w/index.php?oldid=371565325> *Contributors:* Algebraist, Archelon, CBM, D6, DWIII, Drpickem, Giftlite, Gregbard, Grubber, Hans Adler, Ht686rg90, Jabowery, Jorgen W, Jrtayloriv, Kwiki, Mhss, Michael Hardy, Oleg Alexandrov, Pcap, Physis, Ps ttf, Rorro, Salgueiro, Salix alba, Simon12, Spayrard, Swpb, Thehalfone, Trovatore, 16 anonymous edits

Universal algebra *Source:* <http://en.wikipedia.org/w/index.php?oldid=369074809> *Contributors:* ABCD, APH, Aleph4, AlexChurchill, Algebran, Alink, AllS33ingI, Andres, Arthur Rubin, AshtonBenson, AxelBoldt, Bryan Derksen, Chaos, Charles Matthews, D stankov, David Eppstein, Delasz, Dysprosia, EmilJ, Fredrik, GTBacchus, Giftlite, Gvozdet, HStel, Hairu Dude, Hans Adler, Heldergeoovane, Henning Makhholm, Ilovegrouptheory, Isnow, JaGa, Jrtayloriv, Knotwork, Kowey, Linas, Loupeter, Magidin, Michael Hardy, Mindmatrix, Msh210, Nbarth, Nicks221, Oursipan, Pavel Jelinek, Popopp, Revolver, Rgdboer, Rlupsa, Sam Hocevar, Sam Staton, Sanders muc, Sean Tilson, Smmurphy, Spakoj, Template namespace initialisation script, TheSeven, Tkeu, Toby Bartels, Tompw, Twisted86, Wiki alf, Youandme, Zaslav, Zundark, Zvar, 71 anonymous edits

Model theory *Source:* <http://en.wikipedia.org/w/index.php?oldid=368582330> *Contributors:* Algebraist, Ancheta Wis, Archelon, Arthur Rubin, Avaya1, Barkeep, Bcrowell, Bernard the Varanid, CBM, CRGreathouse, CarlHewitt, Chalst, Charles Matthews, Charles Moss, Chinju, Conversion script, Creidieki, Cyrusc, Damian Yerrick, Danadocus, David Eppstein, David Martland, Dcoetzee, DesolateReality, Dessources, Dysprosia, Elwikipedista, EmilJ, EmmetCaulfield, Eric119, Filemon, Floorsheim, Fred ulisses maranhão, Gauge, Giftlite, Gregbard, Groovenstein, Gryakj, Hairu Dude, Halibutt, Hans Adler, Henrygb, Ht686rg90, Iwnbap, James pic, Jason.grossman, Jim O'Donnell, Joejunsun, Jon Awbrey, JonathanZ, Jonsafari, Joriki, Josh Cherry, Joshua Davis, Julian Mendez, Jusdafax, Karl-Henner, Klausness, Lambiam, Linas, Little Mountain 5, Loadmaster, Lottamiata, Loupeter, Lupin, Majorly, MartinHarper, MathMartin, Mdd, Michael Hardy, Mrw7, Msh210, NawlinWiki, Oleg Alexandrov, Ott2, Paine Ellsworth, Paul August, Pavel Jelinek, Payrard, Pcap, Peter M Gerdes, Pi zero, Pomte, Popopp, Pseudonomous, Qwfp, R.e.b., Rcoog, Revolver, Ringspectrum, Rotem Dan, RuM, Ruud Koot, SDaniel, Sam Hocevar, Sapphic, SaxTeacher, Shellgirl, Smimram, Snoyes, Sparky, Spellchecker, The Anome, Thehalfone, Tillmo, Tkuvho, Toby Bartels, Tomaz.slivnik, Tompsci, Trovatore, Valeria.depaiva, Viebel, WAREL, WikiSBTR, Youandme, Zero sharp, Zundark, 84 anonymous edits

Proof theory *Source:* <http://en.wikipedia.org/w/index.php?oldid=368525458> *Contributors:* Arthur Rubin, Brian0918, Bryan Derksen, CBM, CBM2, Cabe6403, Chalst, Charles Matthews, Comiscuous, David Eppstein, Dbtfz, Dicklyon, Dimal25, Dominus, Dysprosia, Gene Nygaard, Giftlite, Gregbard, Hairu Dude, JRB-Europe, JRSpriggs, Jahiegel, Jni, Jorend, Jtauber, Kntg, Krappie, Kumioko, LaForge, Lambiam, Leibniz, Llywrch, Luqui, Magmi, Mannypabla, Markus Krötzsch, MattTait, Mav, Michael Hardy, Msh210, Nortexoid, Number 0, Pj.de.bruin, Porcher, Qxz, Rizome, Rotem Dan, Tbvdn, The Anome, Thisthat12345, Tillmo, Toby Bartels, Tony, Yonaa, Youandme, 47 anonymous edits

Sequent calculus *Source:* <http://en.wikipedia.org/w/index.php?oldid=370175686> *Contributors:* Albertzeyar, Aleph42, Allan McInnes, Angela, Canley, Chalst, Chinju, Clconway, Dbtfz, Dedalus (usurped), DrBob, E-boy, Erik Zachte, Gandalf61, Giftlite, Goodmorningworld, GregorB, Hamaryns, Henning Makhholm, IsleLaMotte, Jamelan, Joriki, Jpbowen, Julian Mendez, Lambiam, Leibniz, Linas, Lregnier, Magnus Bakken, Markus Krötzsch, Mhss, Mkehr, Noamz, Paul Conradi, Physis, Polux2001, Rjwilmsi, Salix alba, Shushruth, Skelta, Sky-surfer, Spayrard, SunCreator, The Anome, Thüringer, Tizio, Tony1, Waldir, Zundark, 36 anonymous edits

Python (programming language) *Source:* <http://en.wikipedia.org/w/index.php?oldid=372113529> *Contributors:* -Barry-, 0x6adb015, 130.94.122.xxx, 207.172.11.xxx, 24.49.40.xxx, 2disbetter, 4th-otaku, A D Monroe III, A plague of rainbows, A. Parrot, A.Ou, ABF, ALBERTMietus, AThing, Abednig, AbstractBeliefs, AdamGomaa, Adammelson, Aeonimitz, Agnistus, AhmadH, Ahoerstermeier, Ahyl1, Aidan W, Aim Here, Aitias, Akhristov, Akuchling, Akulo, Alan Liefting, Alex Heinz, Alex Libman, AlexLibman, Alexandre.tp, Alsh, AmRadioHed, Amakuha, Andre Engels, Andreas Eisele, Andrejj, Andrew Haji, AndrewJlockley, Andy Dingley, Antaeus Feldspar, Anthon.Eff, Anthoniraj, Antonielli, Anupamsr, Apparition11, Aranel, Archer3, Arichnad, Arite, Arkanosis, Arknascar44, Army, Aronzak, Arved, Asqueella, Asuffield, Atheuz, Atomique, Auntof6, Avnit, AxelBoldt, Azimuts43, B7T, BartlebyScriver, Bawolff, Beetle B., Bellhalla, Benjaminhill, Benwing, Bernd vdB, Betterusername, Bevo, Bitmuse, BlindWanderer, Bloodshedder, Blue bear sd, Bmk, Bobo192, Boffob, Bombastrus, Bondates, Boredzo, Borislav, Brainsik, Brainus, Brettlpb, Brion VIBBER, Bryant.cutler, Bsmntbombdood, Bugnot, C. A. Russell, C1932, CFeyecare, CRGreathouse, Cadr, Caim, Cameltrader, Captain panda, Carlj7, Catapult, Cburnett, Cdhitc, Cek, CharlesC, Chconline, Chealer, Cherlin, Chipp, Chmod007, Chowbok, Chri1753, Chris Chittleborough, Chris Roy, Chris the speller, Chrismith, ChuckEsterbrook, Ciphergoth, Cjyprime, Codeczero, Connelly, Conversion script, Coolboy1234, Coolperson3, CorbinSimpson, Cornellier, Ctachme, CubicStar, Cvrebret, Cybercobra, Cyp2il, Cícero, DHowell, Da monster under your bed, Dan Ros, Danakil, Daniel Hn, Daniel.Cardenas, Daniel.ugra, Daverose 33, David Woodward, Davide125, Dazmax, Dcoetzee, DeadEyeArrow, Dejavu12, Den fjättrade ankan, Dhart, Digamma, Disavian, Discospinster, Djmackenzie, Dmeranda, Docu, Dogocow, Dolcecars, Doldad200, Domtyler, Dotancohen, DouglasGreen, Dougofborg, Dp462090, Draicone, Dreftymac, Drini, Drosboro, Drummle, Dvarrazzo, Dwheeler, Dysprosia, Eagleal, Eastwind, EatMyShortz, EdC, Edaelon, Eddie Parker, Edgeprod, EditAnon, Eduardofov, Edward, Egmontaz, Ehheh, Eivance, Eiwort, El Cubano, Elliskev, Eloquence, Elykyllek, Enchanter, Enochlax, Epl18, Eric B. and Rakim, Erik Zachte, Erikcw, Error, Espen, Etz Haim, EurlEIF, Euynn, Faisal.akeel, Falcorian, Fancypantsss, FatalError, Feedmecereale, Fergofrog, Ferri, Fetofs, Fibonaccii, Finell, FitzSpyder, Flammifier, Flam2006, Flash200, Flatline, Flauto Dolce, Flemnira, Formulax, Frap, Fred Bradstadt, Fredrik, Freqsh0, Freshraisin, Friday13, Friedo, Fubar Obfusco, Furrykef, Fuzzymann, Fvw, Gamma, Garkbit, Garrett Albright, Garzo, Gauss, GeeksHaveFeelings, Generic Player, Gerrit, Gesslein, Giftlite, Gillwill, Gioto, GirsaleDE, Glenn, Gohst, Gokulmadhavan, GoodSirJiva, Goodgerster, Goodone121, Gortsack, Graveenib, GreatWhiteNortherner, GreyTeardrop, Gronky, Guaka, Gutworth, H3h, H3llbringer, Habi, Hairu Dude, HalfShadow, Halo, Hannes Röst, Hansamurai, Hao2lian, Happenstance, Harmil, Hawaian717, Hdante, HrebanHammerTime, HeikoEvermann, HelgeStenstrom, Herbee, Hervegirod, Hfastedge, Hom sepanta, Honeyman, Howcheng, HumbertoDiogenes, Hydrargyrum, Hypo, I80and, IW.HG, Iani, Ianozsvald, Idono, Ignacioerrico, Ilya, ImperatorPlinius, Imz, Inquam, Intgr, Iph, Islanes, Itai, J E Bailey, J.delanoy, JLaTondre, JWB, JYUyang, Jacob.jose, Jed S, Jeltz, JeremyA, Jerome Charles Potts, Jerub, Jhlyton, Jin, Jkpl187, Jf23, Jlin, JoaquinFerrero, Joe Chill, Joggleran, JohnElder, JohnWhitlock, Johnunig, Jonik, Jorge Stolfi, Josh Parris, Josh the Nerd, JoshDuffMan, Joytex, Jsginther, Julesd, KDesk, Karih, Karl Dickman, Karol Langner, Katanzag, Kaustuv, Kazuo Moriwaka, Khb3rd, Kbk, Kbrose, Kenliu, Kesla, Kim Bruning, KimDabelsteinPetersen, King of Hearts, Kkinder, Kl4m, Kl4m-AWB, Kneiphof, Koavf, KodiakBjörn, Korpis, Kozuch, Kris Schnee, Ksn, Kubanczyk, KumpelBert, Kuralyov, Kızılsungur, LFaraone, Lacker, Lambiam, LauriO, Laurusnobilis, Leafman, LeeHunter, Legoktm, Levin, LiDaobing, Lightst, Lino Mastrodomenico, LittleDan, Llamadog903, Loggie, LoveEncounterFlow, Lulu of the Lotus-Eaters, LunaticFringe, Lysander89, MER-C, MIT Trekkie, MMuzammils, Mac, Mach5, Macrocoders, Madmardigan53, Maduskis, Malleus Fatuorum, Marco42, Mark Krueger, Mark Russell, Marko75, Martin.komunide.com, MartinSpacek, Martinkunev, Martinultima, Massysett, Masud1011, Mathnerd314, Matt Crypto, Matthew Woodcraft, MatthewRayfield, Matusu, McSly, Mcherm, Mdd, Meduz, Memty, MesserWoland, Michal Nebyla, Midnight Madness, Mignon, Mikco, Mike Lin, Miles, Minghong, Mipadi, MisterSheik, Mjb, Mjpieters, MkClark, Modster, Moshezadka, Mpradeep, Mr Minchin, MrOllie, MrStalker, Mrsatori, Mtrinke, MuffinFlavored, Murrmur, Mykhal, NanoTy, Nanowolf, Nanshu, NapoliRoma, NathanBeach, Natrius, Ncmathsadist, Nealmb, Nearfar, Neilc, Netoholic, Neuralwiki, Nevvar stark, NevilleDNZ, Nguyen Thanh Quang, Nick Garvey, Nikai, Ninly, Nir Soffer, Nknight, Nkour, Noamraph, Noldoaran, Nono64, Northgrove, NotAbel, Nummify, Nzk, Obradovic Goran, Oda Mari, Oefe, Ohnoitsjamie, Olathe, OlavN, Oli Filth, Oliverlewis, Oneirist, Ossiemanners, Ozzmosis, Polyglut, PHaze, Paddy3118, Palfrey, Papna, Patrickjamesmiller, Pcb21, Peak, Pelago, Pengo, Penguinzig, Pepr, Perfecto, PeterReid, Peterhi, Peterl, Peteturtle, Pharos, Philip Trueman, PhilipR, Phoe6, Physicistjedi, Phædrus, Piet Delport, Pillefj, Pmlineditor, Pointillist, Poor Yorick, Poromenos, Positron, Prolog, ProvelT, Prty, Pyritie, Python.tw, Pythonbook, Quadra23, Quartz25, QuicksilverJohny, QuicksilverPhoenix, Qwfp, RP459, Radagast83, Radenski, Radon210, Raghuraman.b, RazielZero, Rbonvall, Rdhettenger, Reb42, Recent Runes, RedWolf, Rich Farmbrough, RichNk, Richard001, Rjukan, Rjwilmsi, Rmt2m, Robinw77, Rodrigostraus, Rogper, Roland2, Ronyclau, Rsoccl, Rspeer, Rufous, Rummye, Rursus, Ruud Koot, S.Örvarr, S, SF007, Sabre23t, Sachavdk, Salmar, Sam Korn, Sam Pointon, Samohyl Jan, Samuel, Samuel Grant, SandManMattSH, Sander Säde, Sanxiyn, SardonickRic, Sarefo, Sbandrews, SciberDoc, Scorchsaber, Sealcian, Sebb, Seidenstudd, Sen Mon, Senordefarge, SergeyLitvinov, Serprex, Sgeo, Shadownahward, Shervinafshar, Shimei, Shmorhay, Sidecharm10101, Sietse Snel, Silverfish, Simeon, Simetrical, Simxp, Sir Isaac, SkyWalker, Sltchfield, Smjg, Snori, Softarch Jon, Sooperman23, SophisticatedPrimate, Spaceygyu, Spebi, SpencerWilson, Squilibob, Sridharinfinity, Ssd, Stangaa, Starnestommy, Stephen Gilbert, Steveha, Strombrg, StuartBrady, Sturdyby, Sullan, Sverdupr, Swaroopch, TJRC, TxiKi, TakuyaMurata, Taniquetl, Tarquin, Technobadger, Teehee123, Telemakh0s, Template namespace initialisation script, Tennessee, Thatos, The Anome, The best jere, TheDolphin, TheParanoidOne, Thebrid, Thejapaneseeek, Thing, Thumpervard, Tide rolls, TimTay, Timmorgan, Timothy Clemans, Timwi, Tlesher, Tobias Bergemann, Tokigun, Tompsci, Tony Sidaway, Tony1, TonyClarke, Toussaint, Towel401, Trampolineboy, Traxs7, Troeger,

Trogdor31, Tualha, TuukkaH, Uniguxy, Unyoyega, UserVOBO, Uyuyuy99, VX, Verdlanco, Visor, V1hurg, Vorratt, Vsion, Wangi, WatchAndObserve, Wavelength, Weel, Wellington, WhatisFeelings?, Whowhat16, Wikiborg, Wiretse, Wlievens, WoLpH, Wrathchild, Wrp103, Ww, Wws, X96lee15, XDanieltx, Xasxas256, Xaxafrad, Xdcxh, Yamaplos, Yath, Yellowstone6, Ygramul, Yoghurt, Yoric, Ytyoun, Yworo, ZeroOne, Zoicon5, Zondor, Zukeper, Zundark, Цика Пика, ۱۷۳۵, 1143 anonymous edits

Sage (mathematics software) *Source:* <http://en.wikipedia.org/w/index.php?oldid=370239636> *Contributors:* Aarktica, Agc, AlmostReadytoFly, Angry bee, AstronomyObserver, Baccyak4H, Balabiot, Benjaminnevans82, Bkell, Bsilverthorn, CRGreathouse, Caco de vidro, ChrisEich, Cjfsyntropy, Cloudruns, Cole Kitchen, Cybercobra, DStoykov, DavidJoynr, DeluxNate, Den fjättrade ankan, Dino, Disavian, Drkirkby, Dwheeler, Elvenlord Elrond, Flamingspinach, Fleetcommand, Frap, Free Software Knight, Fripp, Galoisgroupie, Gesslein, Giftlite, Gihanuk, Gioto, Glennklockwood, GreyTeardrop, Gutworth, Haraldschilly, Herbee, Hottscubbard, Isilanes, JLaTondre, Karnesky, Kay Dekker, Lambiam, Lightmouse, LinguistAtLarge, Masगतotkaca, Maury Markowitz, Maxal, Melnaakeeb, Mietchen, Mr.K., Mschu, Muddl, Muro de Aguas, MySchizoBuddy, Mykhal, Nasa-verve, Neilc, Nv8200p, Ondra.pelech, Ovis23, Penfold1000, Peterih, Phatsphere, Phil, Piccolist, Pleasantville, Pythonicsnake, RDBury, ReTrY, Rich Farmbrough, Rjwilmsi, Robertwb, Rwww, Scooty, Staecker, T68492, Taxman, Thoughtmeister, Throwawayhack, Thumperward, Timothy Clemans, Tobias Bergemann, Tuxcantfly, Uliba, VladimirReshetnikov, Vy0123, 111 anonymous edits

Decision problem *Source:* <http://en.wikipedia.org/w/index.php?oldid=350829942> *Contributors:* 128.138.87.xxx, Arthens, Ascánder, AxelBoldt, Baiji, CBM, CRGreathouse, Chinju, Conversion script, Creidieki, Culix, Cwitty, David.Monniaux, Dcoetzee, Dlakavi, Drae, Dratman, Ehn, Eiji adachi, Gdr, Giftlite, Giraffedata, Gregbard, Hadal, Inklings, Iseeaboar, Jafet, Jonathan de Boyne Pollard, Kesla, Kragen, Krymson, Kurykh, Kuszi, LC, Lalaith, Lambiam, MIT Trekkie, Mandarax, MathMartin, Mellum, MementoVivere, NekoDaemon, Noroton, Nortexoid, Obradovic Goran, Od Mishehu, Oleg Alexandrov, Pakaran, Paul August, Pcap, Philpp, Picpich, Pro8, RobinK, Salsa Shark, SalvNaut, Seb, Sho Uemura, Shreevatsa, Sligocki, Stevertigo, UberScienceNerd, Uday, Unixxx, UsaSatsui, Wavelength, Wbaily, Ylloh, 36 anonymous edits

Boolean satisfiability problem *Source:* <http://en.wikipedia.org/w/index.php?oldid=371114588> *Contributors:* 151.99.218.xxx, Ahalwright, Alex R S, AndrewHowse, Ap, Artem M. Pelenitsyn, B4hand, Bovineboy2008, Brion VIBBER, CBM, CRGreathouse, Chalst, ChangChienFu, Conversion script, Creidieki, DBeyer, DFRussia, Damian Yerrick, David.Monniaux, Dcoetzee, Doomdayx, Dbrzenjev, Dwheeler, Dysprosia, Elias, Enmiles, Everyking, Fancieryu, FlashSheridan, Fratrep, Gdr, Giftlite, Gregbard, Guarani.py, Gwern, Hans Adler, Hattes, Hh, Igor Markov, J. Finkelstein, Jan Hidders, Janto, Jecar, Jimbreed, Jok2000, Jon Awbrey, Jpbowen, Julian Mendez, Karada, Karl-Henner, LC, Leibniz, Localzuk, LouScheffer, Luuva, Magnus, Maming, MathMartin, Max613, McCart42, Mellum, Mets501, Mhym, Michael Hardy, Michael Shields, Miyim, Mjuaez, Mmn100, Mqasem, Mutilin, Naddy, Night Gyr, Nilmerg, Obradovic Goran, Oerjan, Oliver Kullmann, PoeticVerse, PsyberS, Quaternion, RG2, Saforrest, Sam Hooever, Schneelocke, Siteswapper, Sl, Tim Starling, Timwi, Tizio, Twanvl, Twiki-walsh, Vegasprof, Weichaoliu, Wik, Wbaily, Yuri.chebiryak, Yworo, Zander, Zarrapastro, Zeno Gantner, ZeroOne, Станислав, 122 anonymous edits

Constraint satisfaction *Source:* <http://en.wikipedia.org/w/index.php?oldid=370049676> *Contributors:* AndrewHowse, Antonielly, Auntof6, Carbo1200, D6, Deflective, Delirium, Diego Moya, EagleFan, EncMstr, Epktsang, Ertuocel, Grafen, Harburg, Jdpipe, LizBlankenship, MilFlyboy, Nabeth, Ott2, Radsz, Tgdwyer, That Guy, From That Show!, Timwi, Tizio, Uncle G, Vuara, 22 anonymous edits

Rewriting *Source:* <http://en.wikipedia.org/w/index.php?oldid=359012309> *Contributors:* 16@r, Ancheta Wis, Antonino feitosa, Atoll, BillFlis, CRGreathouse, Charles Matthews, Dcoetzee, Dominus, Dysprosia, EdJohnston, Euyyn, Ezrakilty, Flashmob, Furrykef, Greenrd, H.Marxen, Hippie Metalhead, Itai, Joriki, Jpbowen, Linas, Mets501, Michael Hardy, Mikeblas, NekoDaemon, Oleg Alexandrov, Pcap, Pumpapa, Qwertyus, Rodrigo braz, Sigmundur, Simetrical, Spayard, Stephan Schulz, TakuyaMurata, Thv, Wolfkeeper, 26 anonymous edits

Maude system *Source:* <http://en.wikipedia.org/w/index.php?oldid=367841751> *Contributors:* 0goodiegoodie0, A930913, Acipsen, Alexei Kopylov, Benbread, Bjarne, CRGreathouse, Ceefour, Chalst, Cybertron3, Greenrd, Gwern, Jag2kn, Maneesh, Marknew, Mike.lifeguard, Mipadi, Pcap, Saviourmachine, Taemyr, TheParanoidOne, 22 anonymous edits

Resolution (logic) *Source:* <http://en.wikipedia.org/w/index.php?oldid=367133483> *Contributors:* Acipsen, Antonielly, Barnardb, CBM, Ceilican, Charles Matthews, Cloudyed, DVD R W, EatMyShortz, Ekspiulo, Esowteric, Fresheneesz, Gaius Cornelius, Gregbard, GregorB, Gubbubu, Hannes Eder, Henning Makhholm, Hquain, Irbisgreif, JLD, Jiy, Jorend, Jpbowen, Kku, Kmweber, Llchn, Lagenar, Lifetime, Matt Kovacs, Methossant, Mikolasj, MoraSique, Mu, Mukerjee, N4nojohn, Ogai, P Carn, Pdbogen, Peter M Gerdes, Pgr94, Porphyrus, RatnimSnave, Ruud Koot, Simeon, Stephan Schulz, SwordAngel, Slawomir Bialy, T boyd, Termar, Thaukko, Tizio, Todds1, Xitdedragon, Zero sharp, 44 anonymous edits

Automated theorem proving *Source:* <http://en.wikipedia.org/w/index.php?oldid=370445938> *Contributors:* Adrianwn, Agent1, Akriasas, Alai, Algebraist, Ancheta Wis, Antonielly, Arapajoe, Arthur Rubin, AshtonBenson, AxelBoldt, BSoD, Beland, Bfinn, Bobblewik, Bracton, Brian Huffman, Brighterorange, CBM, CRGreathouse, CarolGray, Charles Matthews, Clonway, Conversion script, Cwitty, DaYZman, DainDwarf, Dcoetzee, Detla, Doezyxty, Dwheeler, Dysprosia, Ezrakilty, Geoffgeoffgeoff3, Giftlite, Grafen, GreyCat, Haberg, Iisluacs, JHAlcomb, JYUyang, JakeVortex, Jimbreed, JohnAspinall, Jon Awbrey, JosefUrban, Jpbowen, Jrtayloriv, Karada, Krischik, Ksyrie, Lancebledsoe, Laurusnobilis, Loadmaster, Logperson, Lumingz, LunaticFringe, MaD70, McGeddon, Michael Bednarek, Michael Hardy, Michaeln, MovGP0, Nahaj, Nattfodd, Nealcadwell, Nehalem, Newbygueses, Oleg Alexandrov, Paul Haroun, PaulBrinkley, Peter M Gerdes, PhS, Pintman, Pontus, Qwertyus, Rama, Ramykaram, Rjwilmsi, Rotem Dan, Saeed Jahed, Slawekk, Stephan Schulz, Tatzelworm, Taw, Tc, The Anome, TheosThree, Tim32, TittoAssini, Tizio, Tobias Bergemann, Uwe Bubeck, Wikinglouis, Zacchiro, Zmoboros, Zvika, 152 anonymous edits

Prover9 *Source:* <http://en.wikipedia.org/w/index.php?oldid=315073595> *Contributors:* Dwheeler, Kariteh, Michael Kinyon, Pcap, Tobias Bergemann, 1 anonymous edits

Method of analytic tableaux *Source:* <http://en.wikipedia.org/w/index.php?oldid=369570540> *Contributors:* ArthurDenture, BrideOfKripkenstein, CBM, CBM2, Cetinsert, Chalst, Charles Matthews, DHGarrette, DRap, Dbtzf, Giraffedata, Gregbard, Hekanibru, Hilverd, Kmicinski, LAX, Mets501, Michael Devore, Michael Hardy, Morven, Nlu, Nortexoid, Oleg Alexandrov, Paul Conradi, Rjwilmsi, RobHar, S. Neuman, Salix alba, Sam Staton, Saurc zlunga, Stephan Schulz, Swartik, Tillmo, Tizio, WereSpielChequers, WillWare, XDanieltx, 54 anonymous edits

Natural deduction *Source:* <http://en.wikipedia.org/w/index.php?oldid=371682652> *Contributors:* Aij, Aleph42, Alikhtarov, Ansa211, BD2412, Benwing, CBM2, Chalst, Charles Matthews, Chinju, Cholling, Clemente, Cmdrjameson, Comicist, Cybercobra, Dbtzf, Dgies, EdH, Givegains, Glenn Willen, Gregbard, Hairy Dude, Hugo Herbelin, Iridescent, Jeltz, Jleedev, Joriki, Kaustuv, Lipedia, Marc Mongenet, Mate Juhasz, Mhss, Michael Hardy, Oterhaar, Phil Boswell, Physis, Rowandavies, Rspeer, That Guy, From That Show!, The Cuncator, TheDean, Valeria.depaiva, Vesa Linja-aho, Waldir, Welshbyite, William Lovas, 72 anonymous edits

Isabelle (theorem prover) *Source:* <http://en.wikipedia.org/w/index.php?oldid=370242889> *Contributors:* AN(Ger), Adrianwn, Alain345, Alex Krauss, Ascánder, BillFlis, Caesura, Cek, Cwitty, David Haslam, Dcoetzee, Den fjättrade ankan, Drevicko, DvO, Everyking, Frap, GerwinKlein, Grafen, Greenrd, Gwern, Ixf64, Jpbowen, Lambiam, Michael Hardy, RJFJR, Rich Farmbrough, Slawekk, Spayard, Tbleher, Tillmo, Tobias Bergemann, Urhixidur, VictorPorton, Vkcuncak, Zeno Gantner, Zoonfaer, 20 anonymous edits

Satisfiability Modulo Theories *Source:* <http://en.wikipedia.org/w/index.php?oldid=365436286> *Contributors:* Auntof6, Backjumper, CBM, Caerwine, Clonway, Companion^3, Esrogs, Igor Markov, Jeff3000, LouScheffer, Malekith, Melchoir, Michael Hardy, Mutilin, Radagast83, Rich Farmbrough, Roberto.bruttomesso, Robthebob, Sheini, Trevor hansen, 47 anonymous edits

Prolog *Source:* <http://en.wikipedia.org/w/index.php?oldid=371754422> *Contributors:* 216.60.221.xxx, AaronSloman, Academic Challenger, Aenar, Agne27, Ahoerstemeier, Albmont, AlistairMcMillan, Alterego, AnarchyElmo, Andreas Kaufmann, Andrejj, Angertigger, AnonMoos, BACbKA, Bfinn, Boleslav Bobcik, BorgQueen, Borislav, Brie0, Brion VIBBER, Cal 1234, Camping Fag, Canderra, CanisRufus, CanHewitt, Cbayly, Cedars, Chalst, Charles Matthews, Chinju, Cholling, Christian List, Chuunen Baka, Closedmouth, Connelly, Conversion script, Curtis, Cwofsheep, Cybercobra, DNewhall, Damian Yerrick, Dan Atkinson, Danakil, Dancraggs, Dardasavta, David.Monniaux, Dcoetzee, DigitalCharacter, Dimitar Dobrev, Dimwell, Dirk Beyer, Dmb000006, Dr Debug, Dysprosia, ESSch, Eaefermov, Earl grey, EatMyShortz, Egmontaz, Elizabeth Safo, Elwikipedista, EngineerScotty, Epideme, Erez urbach, Falcon8765, FatalError, Fieldmethods, Fjarlq, Frank Shearar, Friedo, Fubar Obfusco, Furby100, Furrykef, GRBerry, Geodyde, Giulio.piancastelli, Grafikm fr, Gregbard, Grshiplett, Grunt, Gubbubu, Haakon, Hadal, Haham hanuka, Hairy Dude, Hooperbloob, Hqb, Hyad, Idont Havaname, Ihope127, I hamster, Jacj, James Crippen, JanCux, Jeltz, Johnpool, Kazrak, Kinema, Kingpin13, Klausness, Kmarple1, Komap, Korath, Ksn, Kwertii, LOL, LadyEditor, Ldo, LiDaobing, Locke411, Logperson, Looxix, Luc4, Lukas Mach, Luna Santin, LunaticFringe, Maian, Malcolmx15, Marc omorain, Marekpetrik, MarkWahl, Markus Kuhn, Marudubshinki, Matt.whitby, Mboverload, Memming, Merc, Michael Hardy, Mig21bp, NSR, Nasa-verve, Nemti, Nixdorf, NotInventedHere, Nuno Morgadinho, Nzhou, Ojw, One, Optim, Paddu, PamD, Papppfaffe, ParkerJones2007, Pavel Vozenilek, Pgr94, Phatom87, Philip Trueman, Poor Yorick, Pseudomonas, Q0, Qeny, Quackor, Quamaretto, Qwertyus, Raven in Orbit, Rice Yao, Richard Katz, Risk one, Robert Kowalski, Rpinchbeck, Ruud Koot, Rwww, Rwxrwxrwx, Sam Korn, San taunk, Sander123, Savidan, SchnitzMannGreek, Shadow1, Sharcho, Skarebo, Sketch-The-Fox, Sockpuppet538, Spoon1, Stassats, StaticGull, Stephenw32768, SteveLoughran, Stevietheman, Strangerz, Susvolans, Sweavo, TakuyaMurata, Tannin, Taxelson, Template namespace initialisation script, The Wild Falcon, The undertow, Thomas Linder Puls, Thron7, Thumperward, Thüringer, Timmy12, Tinkar, Tizio, Tobias Bergemann, Tomaxer, Tournesol, Transys, TreasuryTag, Troels Arvin, Twinxor, UU, Ultraexactzz, Untalker, Urhixidur, Uriyan, Venar303, VictorAnyakin, Who, Willcannings, Xezlec, Yaztromo, Zayani, 613 anonymous edits

Image Sources, Licenses and Contributors

Image:Excluded middle proof.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Excluded_middle_proof.png *License:* Public Domain *Contributors:* Paul Brauner

Image:Python logo.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Python_logo.svg *License:* unknown *Contributors:* Artem Karimov, DzinX, Kulshrax, Lulu of the Lotus-Eaters, MesserWoland, Monobi, 2 anonymous edits

File:Wikibooks-logo-en.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Wikibooks-logo-en.svg> *License:* logo *Contributors:* User:Bastique, User:Ramac

Image:Python add5 syntax.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Python_add5_syntax.png *License:* unknown *Contributors:* User:Lulu of the Lotus-Eaters

Image:Sage logo new.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Sage_logo_new.png *License:* Creative Commons Attribution 3.0 *Contributors:* The Sage team

File:Sage - plot.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Sage_-_plot.png *License:* unknown *Contributors:* sage

File:SAGE equation solve.jpeg *Source:* http://en.wikipedia.org/w/index.php?title=File:SAGE_equation_solve.jpeg *License:* GNU Free Documentation License *Contributors:* Kolas

Image:Decision_Problem.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Decision_Problem.png *License:* GNU Free Documentation License *Contributors:* User:RobinK

Image:Partially built tableau.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Partially_built_tableau.svg *License:* GNU Free Documentation License *Contributors:* RobHar, Tizio

Image:Prop-tableau-1.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Prop-tableau-1.svg> *License:* Public Domain *Contributors:* RobHar, Tizio

Image:Prop-tableau-2.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Prop-tableau-2.svg> *License:* Public Domain *Contributors:* RobHar, Tizio

Image:Prop-tableau-3.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Prop-tableau-3.svg> *License:* Creative Commons Attribution-Sharealike 2.5 *Contributors:* RobHar, Tizio

Image:Prop-tableau-4.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Prop-tableau-4.svg> *License:* GNU Free Documentation License *Contributors:* RobHar, Tizio

Image:Non-closed propositional tableau.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Non-closed_propositional_tableau.svg *License:* GNU Free Documentation License *Contributors:* RobHar, Tizio

Image:First-order tableau.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:First-order_tableau.svg *License:* GNU Free Documentation License *Contributors:* RobHar, Tizio

Image:First-order tableau with unification.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:First-order_tableau_with_unification.svg *License:* GNU Free Documentation License *Contributors:* RobHar, Tizio

Image:Search tree of tableau space.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Search_tree_of_tableau_space.svg *License:* GNU Free Documentation License *Contributors:* RobHar, Tizio

File:first order natural deduction.png *Source:* http://en.wikipedia.org/w/index.php?title=File:First_order_natural_deduction.png *License:* GNU Free Documentation License *Contributors:* Darapti, Maksim

License

Creative Commons Attribution-Share Alike 3.0 Unported
<http://creativecommons.org/licenses/by-sa/3.0/>
