CDM

Circuits

Klaus Sutner

Carnegie Mellon University

20-bool-alg 2017/12/15 23:21

1. Boolean Algebras

- Boolean Circuits

- From Circuits to Logic

A Boolean algebra is a structure $\mathcal{B} = \langle B, +, \cdot, \overline{\ }, 0, 1 \rangle$ where the following system of equations (BA) is valid:

$$x + (y + z) = (x + y) + z \qquad x \cdot (y \cdot z) = (x \cdot y) \cdot z$$

$$x + y = y + x \qquad x \cdot y = y \cdot x$$

$$x + 0 = x \qquad x \cdot 1 = x$$

$$x + (y \cdot z) = (x + y) \cdot (x + z) \qquad x \cdot (y + z) = (x \cdot y) + (x \cdot z)$$

$$x + \overline{x} = 1 \qquad x \cdot \overline{x} = 0$$

So we have two commutative monoids that coexist peacefully via distributivity, plus a complementation operation.

There is remarkable symmetry between these equations: interchanging plus and times, as well as 0 and 1, takes the left-hand side to the right, and conversely.

There are even terms that are self-dual: applying the duality operations takes the term to an equivalent one. For example, "at least two out of three" is self-dual:

$$(x \wedge y) \vee (x \wedge z) \vee (y \wedge z)$$

Exercise

Show that a self-dual term must have an odd number of variables.

The trivial one-point model is useless, it is important to check if there are interesting models. Here are two standard models for (BA).

- Truth values
  The Boolean values "true" and "false" together with disjunction, conjunction and negation:
  $$\langle\, \{\mathsf{ff}, \mathsf{tt}\}, \vee, \wedge, \neg, \mathsf{ff}, \mathsf{tt} \,\rangle$$
- Power sets
  The power set of any set, together with union, intersection and complement:
  $$\langle\, \mathfrak{P}(A), \cup, \cap, {}^{-}, \emptyset, A \,\rangle$$

Exercise

Verify that these structures are indeed Boolean algebras.

- Divisor Lattices
  For any positive natural number n let $\mathrm{Div}(n)$ be the set of all divisors of n. Then
  $$\langle\, \mathrm{Div}(n), \mathsf{lcm}, \gcd, n/x, 1, n \,\rangle$$
  is a Boolean algebra provided that n is square-free.

- Finite/cofinite Sets
  Let $\mathfrak{P}'(A)$ be the set of all subsets of A that are either finite or cofinite (the complement is finite).
  $$\langle\, \mathfrak{P}'(\mathbb{N}), \cup, \cap, {}^{-}, \emptyset, \mathbb{N} \,\rangle$$

Exercise

Show that the divisor lattice is a Boolean algebras if, and only if, n is square-free. Do these structure look familiar?

Note that we can associate a partial order with every Boolean algebra:

$$x \leq y \iff x \cdot y = x \iff x + y = y$$

In the standard two-element algebra $\mathbb{B}$ this corresponds just to the usual order ff $<$ tt.

For $\mathfrak{P}(A)$ we have $x \leq y \iff x \subseteq y$.

Conversely, a partial order with the right properties can be used to define a Boolean algebra (using sups and infs).

Arbitrary Boolean algebras are rather complicated objects, but in the finite case it is easy to give a complete description.

Let $\mathcal{B} = \langle B, +, \cdot, ^-, 0, 1 \rangle$ be a Boolean algebra.

Definition (Atoms)

An element $a \in \mathcal{B}$ is an atom if $a \neq 0$ but $x \leq a$ implies $x = 0$ or $x = a$.

A Boolean algebra is atomic if whenever $x \not\leq y$ there is an atom $a \leq x$ but $a \not\leq y$.

A Boolean algebra is atomless if it has no atoms.

For example, in $\mathfrak{P}(A)$ the atoms are the singletons $\{a\}$.

Lemma

Every finite Boolean algebra is atomic.

Proof. Since $x = xy + x\overline{y}$ it follows from $x \not\leq y$ that $x\overline{y} \neq 0$. If $x\overline{y}$ is an atom we are done. Otherwise let a be such that $0 < a < x\overline{y}$. If a is an atom we are done. Otherwise we repeat finding $0 < a' < a$ and so on. The process must terminate since the algebra is finite. □

As a consequence, for every $0 \neq b \in \mathcal{B}$ there exists an atom a such that $a \leq b$.

Theorem

Every finite Boolean algebra $\mathcal{B}$ is isomorphic to a powerset.

Exercise

Prove the theorem.

One might hope that the last theorem also takes care of arbitrary Boolean algebras.

Alas, things become much more complicated and one needs topology to produce good characterizations. If $B \subseteq \mathfrak{P}(A)$ is closed under the operations union, intersection and complement we obtain a Boolean algebra. These algebras are called fields of sets. As it turns out, in essence there are no other Boolean algebras.

Theorem (Stone Representation Theorem, 1936)

Up to isomorphism, every Boolean algebra is a field of sets.

The critical part is to construct the right field of sets for a given Boolean algebra (one uses the clopen sets in the associated Stone space).

To see that the argument for the finite case collapses in the infinite case note that there are atomless Boolean algebras.

Call $A \subseteq \mathbb{N}$ periodic if $a \in A \iff a + p \in A$ for some $p \geq 1$. So $A = A_0 + p\mathbb{N}$ where $A_0 \subseteq \{0, 1, \ldots, p-1\}$. Note that complementation does not change the period and union of A and B produces period at most $\mathrm{lcm}(p, q)$. Hence the collection of all periodic subsets of $\mathbb{N}$ forms a Boolean algebra.

But if $A \neq \emptyset$ then $B = A_0 + 2p\mathbb{N}$ lies strictly between $\emptyset$ and A: the algebra is atomless.

Exercise

How about ultimately periodic sets?

(BA) has many interesting semantic consequences. Some particularly important ones are:

$$\begin{aligned} x + x &= x & x \cdot x &= x \\ x + x \cdot y &= x & x \cdot (x + y) &= x \\ x + 1 &= 1 & x \cdot 0 &= 0 \\ \overline{x + y} &= \overline{x} \cdot \overline{y} & \overline{x \cdot y} &= \overline{x} + \overline{y} \\ \overline{\overline{x}} &= x \end{aligned}$$

It is easy to see that these all hold in the models above, but the point is that they hold in all models of (BA). But showing that turns out to be harder than you might think.

Exercise

Verify that these equations hold in the two standard models.

So how do we go about proving that the Boolean laws have these other equations as consequences?

We have to rely on reasoning about equations. For example, if we know that $s = t$ and $t = u$ hold in a structure, then we can conclude that $s = u$ also holds.

Likewise, we are allowed to substitute arbitrary terms for variables, very much in the way we dealt with associativity above.

The crucial point is that all our manipulations must preserve syntactic consequence. We'll make this more precise in a minute, first some examples.

Claim

$(\mathrm{BA}) \models x + x = x$

Proof.

$$\begin{aligned} x + x &= (x + x) \cdot 1 \\ &= (x + x) \cdot (x + \overline{x}) \\ &= x + x \cdot \overline{x} \\ &= x + 0 \\ &= x \end{aligned}$$

□

Note that the third step uses distributivity of plus over times in the "opposite" direction.

By duality, we also have $xx = x$.

Claim

$(\mathrm{BA}) \models 1 + x = 1$

Proof.

$$1 + x = (x + \overline{x}) + x = (x + x) + \overline{x} = x + \overline{x} = 1$$

□

Claim

$(\mathrm{BA}) \models x + xy = x$

Proof.

$$x + xy = x \cdot (1 + y) = x \cdot 1 = x$$

□

The arguments just given are perfectly correct and could appear in any textbook.

Since we are interested in algorithms, there is a question, though: could we automate these arguments?

More precisely, could we build a theorem prover that, given the axioms, would generate these consequences?

Or, at least, could we build a proof assistant that, given an alleged argument, will check correctness and maybe fill in a few steps here and there?

A binary operation $*$ is associative if it satisfies

$$(x * y) * z = x * (y * z)$$

This is usually explained more casually by saying "parens don't matter": it does not matter where we place the parentheses in an expression involving only $*$: the final result of an evaluation is always the same.

We could essentially flatten out any expression to

$$x_1 * x_2 * x_3 * \ldots * x_n$$

Associative operations abound in algebra, addition and multiplication of various kinds of numbers (naturals, integers, rationals, reals, complexes) or matrices are always associative. Typical associative operations in CS are concatenation of words and join of lists.

The usual counterexample is exponentiation: $2^{3^2} \neq (2^3)^2$.

Pairing operations when implemented by some kind of record data structure also fail to be associative: $\langle a, \langle b, c\rangle\,\rangle$ is usually not the same as $\langle\langle a, b\rangle, c\rangle$ (though they are in some sense "equivalent").

So how do we get from the equation

$$(x * y) * z = x * (y * z)$$

to the assertion "parens don't matter"?

At first glance, this associativity axiom may seem to be rather weak, it only handles expressions with two occurrences of the operation $*$. But how about, say,

$$((a * b) * c) * d \qquad \text{versus} \qquad a * (b * (c * d))$$

The last two expressions are in fact equal by associativity. Exactly how does this follow? More importantly, can we generate such a proof automatically?

Here is a proof that uses only the associativity assumption and some equational reasoning. We need to determine which terms have to be assigned to the variables in the axiom to get the desired effect.

$$
\begin{array}{ll}
((a * b) * c) * d & \\
\quad \Downarrow & a * b/x, c/y, d/z \\
(a * b) * (c * d) & \\
\quad \Downarrow & a/x, b/y, c * d/z \\
a * (b * (c * d)) &
\end{array}
$$

Here $a * b/x$ means: substitute $a * b$ for x and so on.

Hence the application of the given associativity equation requires some sort of pattern matching: we have to determine how to bind the variables to terms.

Emboldened by first success, let's try something more ambitious.

Claim

Associativity implies that left associated multiplication is the same as right associated multiplication:

$$(\ldots(x_1 * x_2) * \ldots * x_{n-1}) * x_n = x_1 * (x_2 * (\ldots(x_{n-1} * x_n)\ldots)$$

Proof.

Note that this is really not just one claim but infinitely many: one for each value of $n \geq 0$; there will induction somewhere.

First we establish a little auxiliary result: we can "pull out" the first term in the left associated product:

$$(\ldots((\mathbf{a} * x_1) * x_2) * \ldots * x_{n-1}) * x_n = \mathbf{a} * ((\ldots(x_1 * x_2) * \ldots * x_{n-1}) * x_n)$$

We prove

$$(\ldots((\mathbf{a} * x_1) * x_2) * \ldots * x_{n-1}) * x_n = \mathbf{a} * ((\ldots(x_1 * x_2) * \ldots * x_{n-1}) * x_n)$$

by induction on n.

$n = 0$ is easy, so assume the claim holds for $n - 1 \geq 0$.

$$((\ldots((\mathbf{a} * x_1) * x_2) * \ldots * x_{n-1}) * x_n =$$

by induction hypothesis

$$(\mathbf{a} * (\ldots(x_1 * x_2) * \ldots * x_{n-1})) * x_n =$$

by associativity

$$\mathbf{a} * ((\ldots(x_1 * x_2) * \ldots * x_{n-1}) * x_n)$$

Case $n = 3$ is just the associativity axiom. So assume we have the claim for $n \geq 3$. Then

$$(\ldots((\mathbf{a} * x_1) * x_2) * \ldots * x_{n-1}) * x_n =$$

by auxiliary claim

$$\mathbf{a} * ((\ldots(x_1 * x_2) * \ldots * x_{n-1}) * x_n) =$$

by IH

$$\mathbf{a} * (x_1 * (x_2 * (\ldots(x_{n-1} * x_n)\ldots))$$

□

It is important to note that each of the formulae

$$\varphi_n : (\ldots(x_1 * x_2) * \ldots * x_{n-1}) * x_n = x_1 * (x_2 * (\ldots(x_{n-1} * x_n)\ldots)$$

can be proven from the associativity axiom, using a separate proof for each n.

We are not currently interested in a proof for $\forall\, n\, \varphi_n$ or some such, this would require a much more powerful system.

And don't even think about forming an infinite conjunction

$$\varphi_0 \wedge \varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_n \wedge \ldots$$

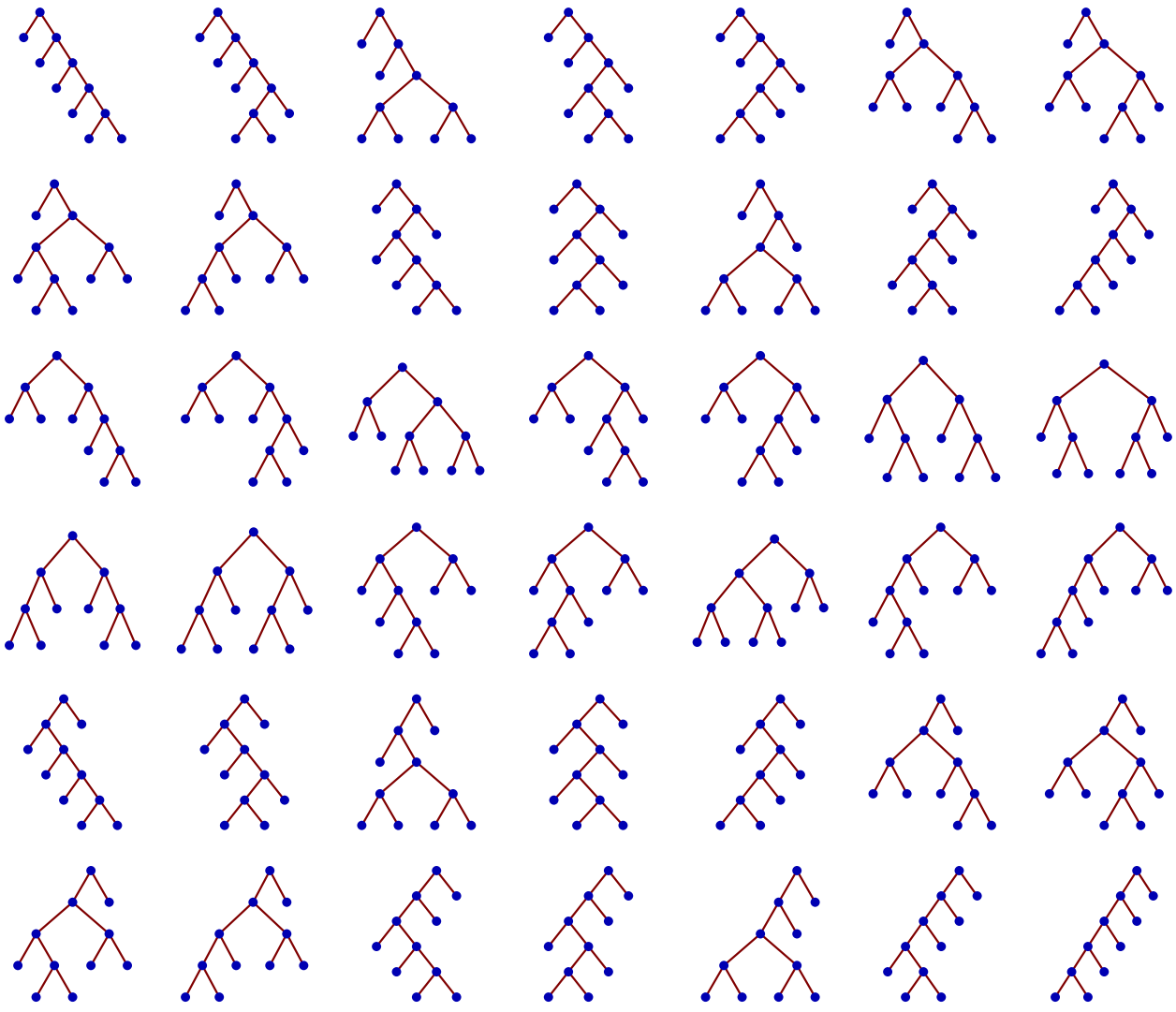
So we have proven that the associativity axiom has the consequence

$$(\ldots(x_1 * x_2) * \ldots * x_{n-1}) * x_n = x_1 * (x_2 * (\ldots(x_{n-1} * x_n)\ldots)$$

Of course, this is just one way to parenthesize these expressions, a more ambitious project would be to show that a similar equation can be established for all possible parenthesizations.

Which brings up the question what exactly is meant by "all possible parenthesizations". A good model is the parse tree of an expression: internal nodes denote on occurrence of $*$ and the leaves are labeled by the arguments: if we read off the frontier of the tree from left to right we get $x_1, x_2, \ldots, x_n$.

Note that these trees are full binary trees. We need to show that the associativity axiom, when construed as an equation on full binary trees, allows us to transform all full binary trees with n leaves into one another.



Here is a formula that can not be derived from associativity:

$$x * y = y * x$$

Operations that satisfy this property are called commutative or Abelian.

Note that in order to show that this formula is not derivable from the associativity axiom it is enough to find just one structure in which associativity holds but this formula is false: the single counterexample shows that φ is not a semantic consequence of the associativity axiom, and so it cannot be derivable either.

Non-commutative associative operations are plentiful: take all words over a two-letter alphabet, say, $\{a, b\}$, with concatenation. Clearly $ab \neq ba$ so commutativity fails.

Matrix multiplication is another classical example.

Commutativity may follow from other properties, though. Recall that a group is Boolean if it satisfies

$$x^2 = 1 \qquad (B)$$

We claim that every Boolean group is also Abelian.

To show: $ab = ba$.

	equation	justification
1	$abab = 1$	substitute ab/x in (B)
2	$aabab = a$	L_a, simplify (1)
3	$bab = a$	substitute a/x in (B), simplify (2)
4	$babb = ab$	R_b, simplify (3)
5	$ba = ab$	substitute b/x in (B), simplify (4)

Equational reasoning in the context of an axiomatic system has become the gold standard in some areas of mathematics, in particular in algebra.

More generally, Bourbaki's work shows clearly that describing mathematical structures in terms of well-chosen axioms is an excellent way to guarantee rigor and precision.

In the future, the set-theoretic approach embraced by Bourbaki may ultimately give way to a more categoric approach, but that will argue even more strongly in favor of equational reasoning.

It was recognized in the 1970s that data structures should be modeled independent from any concrete implementation (abstract data types). In a sense, one can axiomatize (some aspects of certain) data structures.

- Algebraic specification of abstract datatypes via signatures and axiomatic descriptions of the primitive operations:
  - Declaration of used types.
  - Declaration of primitive operations.
  - Axioms specifying behavior of operations.
- The description contains virtually no information about implementation.

The lack of implementation helps to simplify the description and provides a clear interface – which is crucial for reuse and for any attempt at correctness arguments.

Here is an ADT type description of a list data structure based on the primitive operation of prepend. Here is the vocabulary:

$a, b, c, \ldots$	atoms
nil	empty list
$x, y, z, \ldots$	lists
prep(a, x)	prepend atom a to list x (constructor)

Specifications: prepending an atom to nil produces the atom, after a prepend, the list is not empty and there is a unique way to decompose a list produced by repeated prepends:

$$\text{prep}(a, x) \neq \text{nil}$$
$$\text{prep}(a, x) = \text{prep}(b, y) \rightarrow a = b \wedge x = y$$

These are the given elementary properties that an implementation is supposed to satisfy.

Given prepend as a primitive, one can define other operations such as append.

$$\mathsf{app}(a, \mathsf{nil}) = \mathsf{prep}(a, \mathsf{nil})$$
$$\mathsf{app}(a, \mathsf{prep}(b, x)) = \mathsf{prep}(b, \mathsf{app}(a, x))$$

We could also define certain destructor operations:

$\mathsf{head}(\mathsf{prep}(a, x)) = a$	head operation
$\mathsf{tail}(\mathsf{prep}(a, x)) = x$	tail operation

Note that these operations are partial, they make no sense for nil.

Another example of a defined operation: the join of two lists.

$$\begin{aligned} \mathsf{join}(\mathsf{nil}, y) &= y \\ \mathsf{join}(\mathsf{prep}(a, x), y) &= \mathsf{prep}(a, \mathsf{join}(x, y)) \end{aligned}$$

This definition will then guarantee certain properties such as associativity.

Note that the induction used easily translates into a recursive algorithm. Of course, in the real world one has to worry about efficiency issues.

One last example, the reversal operation.

$$\mathsf{rev}(\mathsf{nil}) = \mathsf{nil}$$
$$\mathsf{rev}(\mathsf{prep}(a, x)) = \mathsf{app}(\mathsf{rev}(x), a)$$

We claim that given these specifications it follows that

$$\mathsf{rev}(\mathsf{join}(x, y)) = \mathsf{join}(\mathsf{rev}(y), \mathsf{rev}(x)).$$

Apart from induction, the proof is equational.

Exercise

Show that join *is an associative operation.*

Exercise

Prove the join-reversal property.

Exercise

How would you go about adding a predicate `empty` *to this system?*

Exercise

How would you go about adding a Cartesian product operation? This would require nested lists.

Exercise

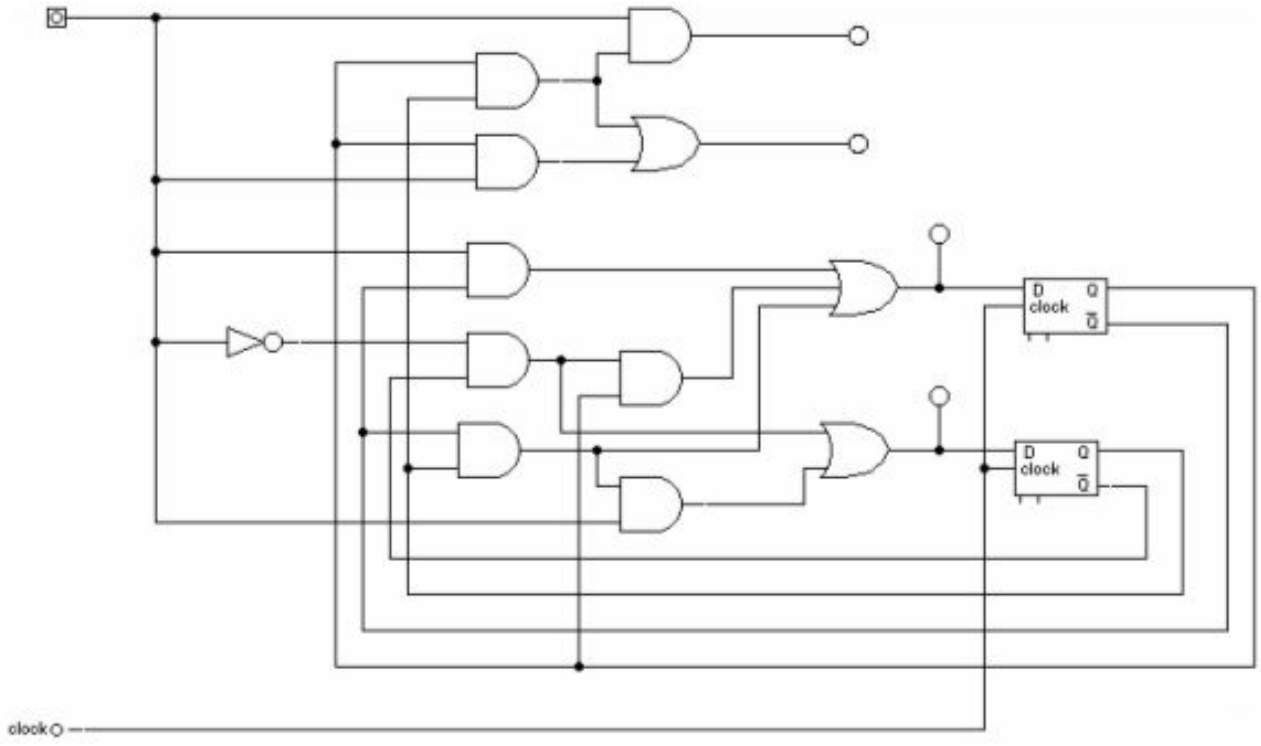
Define a similar definition for binary trees.

- Boolean Algebras

2. Boolean Circuits

- From Circuits to Logic

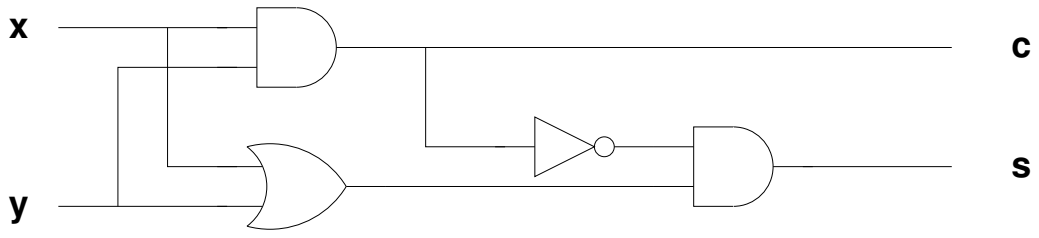
There is a close connection between propositional logic and certain digital circuits where two physical states (such as high/low voltage) can be interpreted as representing truth values.



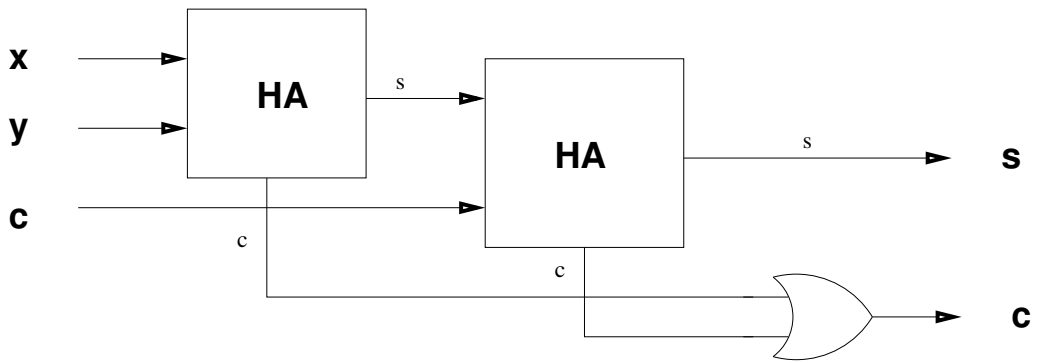
This one has feedback, and is a bit too complicated for our purposes.

Back to basics: consider only feedback-free circuits.

Half Adder



2-bit Adder



More precisely, we can formally define these circuits as follows:

- We have an acyclic digraph $G = \langle V, E \rangle$ that has several nodes of in-degree 0 (input nodes) and nodes of out-degree 0 (output nodes). The underlying ugraph is connected.
- The internal nodes of G are labeled by Boolean operators *and*, *or* and *not*. The indegree (here usually referred to as fan-in) of these nodes has to be appropriate.

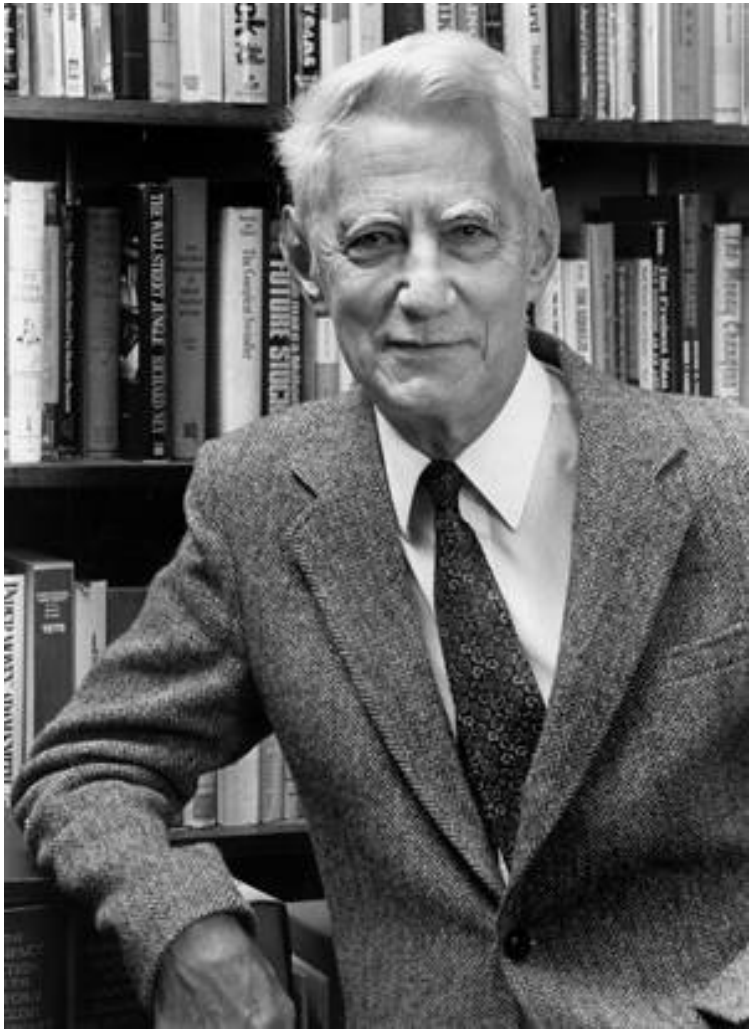
One can limit the fan-in to 2 for *and* and *or* nodes, or allow unlimited fan-in. Negation requires fan-in 1.

For the half adder, we have

$$\begin{aligned} c &= x \wedge y \\ s &= \neg c \wedge (x \vee y) \\ &= x \oplus y \end{aligned}$$

Here we have used an algebraic representation, equality really corresponds to equivalence.

This notation turns out to be quite useful for hand-computations.



The connection between electronic circuits and logic (in the guise of Boolean algebras) was first clearly recognized by Shannon in 1938.

The obvious question arises: what is the underlying algebra?

Definition

A Boolean algebra is a structure $\mathcal{B} = \langle B, +, \cdot, \overline{\ }, 0, 1 \rangle$ of signature $(2, 2, 1, 0, 0)$ where the following equational axioms (BA) are valid:

$$x + (y + z) = (x + y) + z \qquad x \cdot (y \cdot z) = (x \cdot y) \cdot z$$

$$x + y = y + x \qquad x \cdot y = y \cdot x$$

$$x + 0 = x \qquad x \cdot 1 = x$$

$$x + (y \cdot z) = (x + y) \cdot (x + z) \qquad x \cdot (y + z) = (x \cdot y) + (x \cdot z)$$

$$x + \overline{x} = 1 \qquad x \cdot \overline{x} = 0$$

So we have two commutative monoids that coexist peacefully, plus some sort of negation. Note the duality between plus and times in (BA).

Here are two standard models for (BA).

- Truth values
  The Boolean values "true" and "false", written tt and ff below, together with disjunction, conjunction and negation:

$$\langle \{\mathsf{ff}, \mathsf{tt}\}, \vee, \wedge, \neg, \mathsf{ff}, \mathsf{tt} \rangle$$

- Power sets
  The power set of any set, together with union, intersection and complement:

$$\langle \mathfrak{P}(A), \cup, \cap, {}^{-}, \emptyset, A \rangle$$

Exercise

Verify that these structures are indeed Boolean algebras.

- Finite/cofinite Sets
  Let $\mathfrak{P}_\omega(A)$ be the set of all subsets of A that are either finite or cofinite (the complement is finite).

$$\langle\, \mathfrak{P}_\omega(\mathbb{N}), \cup, \cap, {}^{-}, \emptyset, \mathbb{N} \,\rangle$$

- Divisor Lattices
  For any positive natural number n let $\mathsf{Div}(n)$ be the set of all divisors of n. Then

$$\langle\, \mathsf{Div}(n), \mathsf{lcm}, \gcd, n/x, 1, n \,\rangle$$

  is a Boolean algebra provided that n is square-free.

Exercise

Show that the divisor lattice is a Boolean algebras if, and only if, n is square-free. Do these structures look familiar?

(BA) has many interesting semantic consequences. Some particularly important ones are:

$$x + x = x \qquad x \cdot x = x$$

$$x + x \cdot y = x \qquad x \cdot (x + y) = x$$

$$x + 1 = 1 \qquad x \cdot 0 = 0$$

$$\overline{x + y} = \overline{x} \cdot \overline{y} \qquad \overline{x \cdot y} = \overline{x} + \overline{y}$$

$$\overline{\overline{x}} = x$$

It is easy to see that these all hold in the models above, but the point is that they hold in all models of (BA). Alas, showing that turns out to be harder than you might think.

Exercise

Verify that these equations hold in the two standard models.

To model circuits we use the two-element Boolean algebra $\mathbb{B} = \{\mathsf{ff}, \mathsf{tt}\}$. For the sake of readability we often use 0 for false, and 1 for true.

Definition

A Boolean function is a map of the form $\mathbb{B}^n \to \mathbb{B}$. where $n \geq 0$.

Note that we only consider single outputs here

$$f : \mathbb{B}^n \to \mathbb{B}$$

rather than

$$f : \mathbb{B}^n \to \mathbb{B}^m$$

There is no problem with this, we can always use vectors of Boolean functions if multiple outputs are needed.

What kind of Boolean functions are there for small values of n?

Case $n = 0$: 2 functions

All we get is 2 constant functions $\mathbf{0}$ and $\mathbf{1}$.

Case $n = 1$: 4 functions

2 constants, plus

- $H_{\mathsf{id}} = x$, the identity, and
- $H_{\mathsf{not}}(x) = 1 - x$, negation.

Incidentally, the latter two are reversible.

Case $n = 2$ much more interesting: 16 functions.

Some correspond to the typical gates used in circuits.

x	y	$H_{\mathsf{and}}(x, y)$	$H_{\mathsf{or}}(x, y)$	$H_{\mathsf{imp}}(x, y)$
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	1

x	y	$H_{\mathsf{equ}}(x, y)$	$H_{\mathsf{xor}}(x, y)$	$H_{\mathsf{nand}}(x, y)$
0	0	1	0	1
0	1	0	1	1
1	0	0	1	1
1	1	1	0	0

We will see later that all Boolean functions can be constructed from these.

We can write circuits as compositions of Boolean functions.

For the half-adder from above we have:

$$c = H_{\mathsf{and}}(x, y)$$

and

$$s = H_{\mathsf{and}}(H_{\mathsf{not}}(H_{\mathsf{and}}(x, y)), H_{\mathsf{or}}(x, y))$$

This may not look particularly impressive, but the important point is that we are moving closer to algebra.

To keep things close to ordinary algebraic notation we can write

- $+$ for H_{or},
- $\cdot$ for H_{and}, and
- $\overline{x}$ for H_{not}.

But then a digital circuit is just a term $t \in \mathcal{T}(+, \cdot, \overline{\ })$ where the inputs correspond to the free variables.

We are interested in smaller/simpler terms t' such that $t = t'$ and we can use the machinery of Boolean algebra to establish such equalities.

We can use Boolean algebra to construct an actual circuit. Here inputs are x, y and z and the outputs are s (sum) and c (carry). The table describes the functionality of the circuit.

x	y	z	s	c
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

From the table we can read off immediately that

$$s = \overline{x}\,\overline{y}z + \overline{x}y\overline{z} + x\overline{y}\,\overline{z} + xyz$$

$$c = \overline{x}yz + x\overline{y}z + xy\overline{z} + xyz$$

These expressions are mildly complicated, but we can simplify them considerably using equational reasoning in Boolean algebra.

To simplify, let

$$u = x\overline{y} + \overline{x}y = (x + y)\ \overline{xy}$$

then

$$s = \overline{u}z + u\overline{z}$$
$$c = uz + xy$$

Not too bad. u is essentially the half-adder from above, and the last two formulae represent the 2-bit adder.

Note that this simplification process would be rather difficult using circuit diagrams!

The design of he following treatise is to investigate the fundamental laws of the operations of the mind by which reasoning is performed; to give expression to them in the symbolical language of a calculus, and to open this foundation to establish the science of logic and construct its method.

1847 The Mathematical Analysis of Logic

1854 An Investigation of the Laws of Thought

Note that some of laws of Boolean algebra are intuitively easy to grasp, in particular the ones that coexist peacefully with ordinary arithmetic.

Alas, the logical laws that clash with the laws of numbers are far more problematic and take much effort to get used to. It may come as a relief to you that even G. Boole had problems with his very own algebra.

For example, here is an argument he proposed that has some major problems: assuming $F(x) = 0$, show that $F(0)F(1) = 0$. The first step is to use what is now often and inaccurately called Shannon expansion to argue

$$F(x) = F(1)x + F(0)(1 - x) = 0$$

A little rearrangement then yields

$$(F(1) - F(0))x + F(0) = 0$$

Then we can conclude

$$x = \frac{F(0)}{F(0) - F(1)}$$

and hence

$$1 - x = -\frac{F(1)}{F(0) - F(1)}$$

Substituting into $x(1 - x) = 0$ we get

$$0 = -\frac{F(0)F(1)}{(F(0) - F(1))^2}$$

and the claim "follows."

Exercise

Discuss the merits of this argument.

- Boolean Algebras
- Boolean Circuits

3. From Circuits to Logic

To keep things simple, we will focus on three types of gates

- unary: negation
- binary: and, or

Since any Boolean function can be constructed from just these (in fact, "negation" and "and" suffices) we do not lose out on any functions, though the circuits may get larger because of our restriction.

Proposition

Given a Boolean circuit C and values for the inputs one can compute the output in linear time.

This works for any reasonable representation of the circuit.

Exercise

Discuss some plausible implementations of circuits and the corresponding evaluation algorithm.

It is customary to refer to feedback-free Boolean circuits as Boolean formulae or Boolean expressions or propositional formulae.

propositional formulae since these expressions can naturally be interpreted as assertions. Note that the human cognitive system has special abilities when it comes to interpreting such formulae. If we capture the logical structure of an argument in terms of propositional formulae one can check the correctness of the argument relatively easily (at least in simple cases).

Example

$$q \wedge (p \rightarrow \neg q) \rightarrow \neg p$$

Think of the interpretation p: field is finite, q: field has characteristic 0.

The big advantage of this way of looking at Boolean functions is that it provides us with a number of possible ways to analyze them.

For example, one naturally would like to know which functions are "true" or "valid". Which are "false" or "contradictory"?

Also, since reasoning (at least in mathematics and the sciences) often proceeds from basic assumptions (axioms) to conclusions by a sequence of inference steps the question arises how we can model this process in terms of Boolean functions.

The first step in this direction is to give a reconstruction of (a small fragment) of ordinary language in the framework of Boolean functions.

We fix a language for propositional formulae that imitates constructs in ordinary language and, of course, in various programming languages. Sorry, Herr Wittgenstein.

$\bot$, $\top$	constants false, true
p, q, r, ...	propositional variables
$\neg$	not
$\wedge$	and, conjunction
$\vee$	or, disjunction
$\rightarrow$	conditional (implies)
$\leftrightarrow$	biconditional (equivalent)

Negation is unary, all the others a binary.

Formulae are constructed from these variables and operators as usual.

So far we only have syntax, we also need to pin down the meaning of a formula.

Definition

A truth assignment is a map that associates a truth value with each variable.

Given a formula φ and a truth assignment σ for all the variables, we can determine the truth value of φ under α, written $[\![\varphi]\!]_\sigma$.

$$
\begin{aligned}
[\![\bot]\!]_\sigma &= 0 \\
[\![\top]\!]_\sigma &= 1 \\
[\![\neg\varphi]\!]_\sigma &= H_{\mathsf{not}}([\![\varphi]\!]_\sigma) \\
[\![\varphi \vee \psi]\!]_\sigma &= H_{\mathsf{or}}([\![\varphi]\!]_\sigma, [\![\psi]\!]_\sigma) \\
[\![\varphi \wedge \psi]\!]_\sigma &= H_{\mathsf{and}}([\![\varphi]\!]_\sigma, [\![\psi]\!]_\sigma) \\
[\![\varphi \to \psi]\!]_\sigma &= H_{\mathsf{imp}}([\![\varphi]\!]_\sigma, [\![\psi]\!]_\sigma) \\
[\![\varphi \leftrightarrow \psi]\!]_\sigma &= H_{\mathsf{equ}}([\![\varphi]\!]_\sigma, [\![\psi]\!]_\sigma)
\end{aligned}
$$

Of course, $[\![p]\!]_\sigma = \sigma(p)$ is directly given for propositional variables p.

Definition

Let σ be a truth assignment and φ a propositional formula. σ satisfies or models φ if φ is true under σ: $[\![\varphi]\!]_\sigma = 1$.

A truth assignment satisfies a set of formulae Φ if it satisfies each formula in the set.

In symbols

$$\sigma \models \varphi \qquad \text{and} \qquad \sigma \models \Phi$$

Incidentally, this notation is lifted from Frege's Begriffsschrift.

Definition

Let Φ be a set of formulae and φ a propositional formula. φ is a semantic consequence of Φ if for every truth assignment σ that satisfies Φ also satisfies φ: $\sigma \models \Phi$ implies $\sigma \models \varphi$

In symbols,

$$\Phi \models \varphi$$

Example

Here is the famous modus ponens, a classical rule of inference.

$$\varphi, \varphi \to \psi \models \psi.$$

Example

From $\Phi, \varphi \models \bot$ it follows that $\Phi \models \neg\varphi$.

Example

From $\Phi, \varphi \models \psi$ it follows that $\Phi \models \varphi \to \psi$.

Definition

A formula φ is a tautology if it is true under all truth assignments.

In symbols,

$$\models \varphi$$

Tautologies are what corresponds to the informal notion of a "true assertion" or "valid statement" (at least in propositional logic).

Example

$p \vee \neg p$ is the smallest non-trivial tautology

Exercise

Show that the following formula is a tautology:

$$(p \wedge q \to r) \to \big((\neg p \vee q) \to (p \to r)\big)$$

Definition

A formula φ is called a

- contradiction: $\sigma \models \varphi$ for no truth assignment σ
- contingency (or satisfiable): $\sigma \models \varphi$ for some truth assignment σ

Note that φ is a contradiction iff $\neg\varphi$ is a tautology. Likewise, φ is a contingency iff $\neg\varphi$ fails to be a tautology.

Example

$p \wedge \neg p$ is a contradiction.

$q \wedge (p \rightarrow \neg q) \wedge p$ is a contradiction.

More technically, there are some important decision problems associated with our definitions. We would like to solve these problems computationally.

Tautology

- Instance: A propositional formula φ.
- Yes-instance: A tautology.

Satisfiability (decision version)

- Instance: A propositional formula φ.
- Yes-instance: A contingency (i.e., a satisfiable formula).

Satisfiability (search version)

- Instance: A propositional formula φ.
- Solution: A satisfying truth assignment if it exists, "No" otherwise.

Likewise we could introduce a decision problem Contradiction.

They are all closely related, though:

- φ is a tautology iff $\neg\varphi$ is not satisfiable.
- φ is a contradiction iff φ is not satisfiable.

Hence fast (and general, more later) algorithms for one problem could be used to solve the others.

Note, though, that some algorithms require the formula to be given in a special form; negation may clash with this extra condition.

As an example of the expressiveness of SAT, we will show how to translate the Hamiltonian Cycle problem into a satisfiability problem.

Let $G = \langle [n], E \rangle$ be an undirected graph.

We introduce $n(n+1)$ Boolean variables

$$p_{x,t} \qquad 1 \leq x \leq n, 0 \leq t \leq n.$$

The idea is that the Hamiltonian path we are looking for touches node x at time t iff $\sigma(p_{x,t}) = 1$ for a satisfying truth assignment σ.

Let us write EXO$(x_1, \ldots, x_r)$ for a Boolean formula that is true under σ iff exactly one of the variables $x_1, \ldots, x_r$ is true under σ.

We will construct a Boolean formula Φ_G that is satisfiable iff G is Hamiltonian.

The formula is a conjunction with 4 parts as follows:

$$\bigwedge_t \mathsf{EXO}(p_{1,t}, p_{2,t}, \ldots, p_{n,t})$$

$$\bigwedge_{x \neq 1} \mathsf{EXO}(p_{x,1}, p_{x,2}, \ldots, p_{x,n})$$

$$p_{1,0} \wedge p_{1,n}$$

$$\bigwedge_{x,t} p_{x,t} \rightarrow \bigvee_{y \in \Gamma_x} p_{y,t+1}$$

Here $\Gamma_x = \{\, y \in [n] \mid (x,y) \in E \,\}$ denotes the neighborhood of x in G.

Note that the size of Φ_G is $\Theta(n^3)$ and thus polynomial in the size of the graph; moreover, the construction is in a sense trivial (a very simple program can handle the translation) .

Suppose σ satisfies Φ_G. By part 1 there is a sequence of vertices v_t, $0 \leq t \leq n$: let v_t be the unique x such that $\sigma \models v_{x,t}$.

By part 2 every vertex appears on this list. Also, by part 3, $v_0 = v_n = 1$ so that all other vertices must appear exactly once by counting.

Lastly, by part 4, (v_t, v_{t+1}) is an edge.

Hence G has a Hamiltonian cycle – which can be read off directly from the satisfying truth assignment.

In the opposite direction, suppose G has a Hamiltonian cycle. We may think of this cycle as a sequence v_t, $0 \leq t \leq n$ of vertices where $v_0 = v_n = 1$.

Set $\sigma(p_{x,t}) = 1$ iff $v_t = x$.

It is easy to check that σ satisfies Φ_G.

Unfortunately, there is little hope to find fast algorithms for any of the decision problems above.

Theorem

Satisfiability is $\mathbb{NP}$-complete.

Theorem

Tautology is co-$\mathbb{NP}$-complete.

Put differently, all known algorithms depend to some degree on a brute force enumeration of many of the 2^n possible truth assignments.

But how about simple formulae, formulae that are required to have a particular syntactical structure?