Boolean Formalism and Explanations

Eric C. R. Hehner
University of Toronto

Abstract

Boolean algebra is simpler than number algebra, with applications in programming, circuit design, law, specifications, mathematical proof, and reasoning in any domain. So why is number algebra taught in primary school and used routinely by scientists, engineers, economists, and the general public, while boolean algebra is not taught until university, and not routinely used by anyone? A large part of the answer may be in the terminology of logic, in the symbols used, and in the explanations of boolean algebra found in textbooks. The subject has not yet freed itself from its history and philosophy. This paper points out the problems delaying the acceptance and use of boolean algebra, and suggests some solutions.

Introduction

This is not a mathematically deep talk. It does not contain any new mathematical results. It is about the symbols and notations of boolean algebra, and about the way the subject is explained. It is about education, and about putting boolean algebra into general use and practice. To make the scope clear, by “boolean algebra” I mean the usual algebra whose expressions are boolean, where “boolean” is a type. I mean to include propositional logic and predicate calculus. I shall say “boolean algebra”, “boolean calculus”, or “logic” interchangeably, and call its expressions “boolean expressions”. Analogously, I say “number algebra”, “number calculus”, or “arithmetic” interchangeably, and call its expressions “number expressions”.

Boolean algebra is the basic algebra for much of computer science. Other applications include digital circuit design, law, reasoning about any subject, and any kind of specifications, as well as providing a foundation for all of mathematics. Boolean algebra is inherently simpler than number algebra. There are only two boolean values and a few boolean operators, and they can be explained by a small table. There are infinitely many number values and number operators, and even the simplest, counting, is inductively defined. So why is number algebra taught in primary school, and boolean algebra in university? Why isn't boolean algebra better known, better accepted, and better used?

One reason may be that, although boolean algebra is just as useful as number algebra, it isn't as necessary. Informal methods of reckoning quantity became intolerable several thousand years ago, but we still get along with informal methods of specification, design, and reasoning. Other reasons may be accidents of educational history, and still others may be our continuing mistreatment of boolean algebra.

Historical Perspective

To start to answer these questions, I'm going to look briefly at the history of number algebra. Long after the invention of numbers and arithmetic, quantitative reasoning was still a matter of trial and error, and still conducted in natural language. If a man died leaving his 3 goats and 20 chickens to be divided equally between his 2 sons, and it was agreed that a goat is worth 8 chickens, the solution was determined by iterative approximations, probably using the goats and chickens themselves in the calculation. The arithmetic needed for verification was well understood long before the algebra needed to find a solution.

The advent of algebra provided a more effective way of finding solutions to such problems, but it was a difficult step up in abstraction. The step from constants to variables is as large as the step from chickens to numbers. In English 500 years ago, constants were called "nombers denominate" [concrete numbers], and variables were called "nombers abstracte". Each step in an abstract calculation was accompanied by a concrete justification. For example, we have the Commutative Law [0]:

> When the chekyns of two gentle menne are counted, we may count first the chekyns of the gentylman having fewer chekyns, and after the chekyns of the gentylman having the greater portion. If the nomber of the greater portion be counted first, and then that of the lesser portion, the denomination so determined shall be the same."

This version of the Commutative Law includes an unnecessary case analysis, and it has missed a case: when the two gentlemen have the same number of chickens, it does not say whether the order matters. The Associative Law [0]:

> When thynges to be counted are divided in two partes, and lately are found moare thynges to be counted in the same generall quantitie, it matters not whether the thynges lately added be counted together with the lesser parte or with the greater parte, or that there are severalle partes and the thynges lately added be counted together with any one of them.

One of the simplest, most general laws, sometimes called the Transparency Law, or "substitution of equals for equals",

$$x=y \Rightarrow fx=fy$$

seems to have been discovered a little at a time. Here is one special case [1]:

> "In the firste there appeareth 2 nombers, that is $14x + 15y$ equalle to one nomber, whiche is $71y$. But if you marke them well, you maie see one denomination, on bothe sides of the equation, which never ought to stand. Wherfore abating [subtracting] the lesser, that is $15y$ out of bothe the nombers, there will remain $14x = 56y$ that is, by reduction, $1x = 4y$.
>
> Scholar. I see, you abate $15y$ from them bothe. And then are thei equalle still, seyng thei wer equalle before. According to the thirde common sentence, in the patthewaie: If you abate even [equal] portions, from thynges that bee equalle, the partes that remain shall be equall also.
>
> Master. You doe well remember the firste grounds of this arte."

And then, a paragraph later, another special case:

> "If you adde equalle portions, to thynges that bee equalle, what so amounteth of them shall be equalle."

As you can imagine, the distance from $2x + 3 = 3x + 2$ to $x=1$ was likely to be several pages. The reason for all the discussion in between formulas was that algebra was not yet fully trusted. Algebra replaces meaning with symbol manipulation; the loss of meaning is not easy to accept. The author had to constantly reassure those readers who had not yet freed themselves from thinking about the objects represented by numbers and variables. Those who were skilled in the art of informal reasoning about quantity were convinced that thinking about the objects helps to calculate correctly, because that is how they did it. As with any technological advance, those who are most skilled in the old way are the most reluctant to see it replaced by the new.

Today, of course, we expect a quantitative calculation to be conducted entirely in algebra, without reference to thynges. Although we justify each step in a calculation by reference to an algebraic law, we do not have to justify the laws anymore. We can go farther, faster, more succinctly, and with much greater certainty. The following proof of Wedderburn's Theorem (a finite division ring is a commutative field) is typical of today's algebra; I have taken it from the text used when I studied algebra [2]. You needn't read it; I quote it only so that I can comment on it after.

(*start of typical modern proof*)

> Let D be a finite division ring and let Z be its center. By induction we may assume that any division ring having fewer elements than D is a commutative field.
>
> We first remark that if $a,b\in D$ are such that $b^ta=ab^t$ but $ba\neq ab$ then $b^t\in Z$. For, consider $N(b^t) = \{x\in D \mid b^tx=xb^t\}$. $N(b^t)$ is a subdivision ring of D; if it were not D, by our induction hypothesis, it would be commutative. However, both a and b are in $N(b^t)$ and these do not commute; consequently, $N(b^t)$ is not commutative so must be

all of D. Thus $b^t \in Z$.

Every nonzero element in D has finite order, so some positive power of it falls in Z. Given $w \in D$ let the order of w relative to Z be the smallest positive integer $m(w)$ such that $w^{m(w)} \in Z$. Pick an element a in D but not in Z having minimal possible order relative to Z, and let this order be r. We claim that r is a prime number for if $r=pq$ with $1<p<r$ then a^p is not in Z. Yet $(a^p)^q = a^r \in Z$, implying that a^p has an order relative to Z smaller than that of a.

By the corollary to Lemma 7.9 there is an $x \in D$ such that $xax^{-1} = a^i \neq a$; thus $x^2ax^{-2} = x(xax^{-1})x^{-1} = xa^ix^{-1} = (xax^{-1})^i = (a^i)^i = a^{i^2}$. Similarly, we get $x^{r-1}ax^{-(r-1)} = a^{i^{(r-1)}}$. However, r is a prime number thus by the little Fermat theorem (corollary to Theorem 2.a), $i^{r-1} = 1+ur$, hence $a^{i^{(r-1)}} = a^{1+ur} = aa^{ur} = \lambda a$ where $\lambda = a^{ur} \in Z$. Thus $x^{r-1}a = \lambda azx^{r-1}$. Since $x \notin Z$, by the minimal nature of r, x^{r-1} cannot be in Z. By the remark of the earlier paragraph since $xa \neq ax$, $x^{r-1}a \neq ax^{r-1}$ and so $\lambda \neq 1$. Let $b = x^{r-1}$; thus $bab^{-1} = \lambda a$; consequently, $\lambda^r a^r = (bab^{-1})^r = ba^rb^{-1} = a^r$ since $a^r \in Z$. This relation forces $\lambda^r = 1$.

We claim that if $y \in D$ then whenever $y^r = 1$, then $y = \lambda^i$ for some i, for in the field $Z(y)$ there are at most r roots of the polynomial $u^r - 1$; the elements $1, \lambda, \lambda^2, \ldots, \lambda^{r-1}$ in Z are all distinct since λ is of the prime order r and they already account for r roots of $u^r - 1$ in $Z(y)$, in consequence of which $y = \lambda^i$.

Since $\lambda^r = 1$, $b^r = \lambda^r b^r = (\lambda b)^r = (a^{-1}ba)^r = a^{-1}b^ra$ from which we get $ab^r = b^ra$. Since a commutes with b^r but does not commute with b, by the remark made earlier, b^r must be in Z. By Theorem 7.b the multiplicative group of nonzero elements of Z is cyclic; let $\gamma \in Z$ be a generator. Thus $a^r = \gamma^j$, $b^r = \gamma^k$; if $j = sr$ then $a^r = \gamma^{sr}$; whence $(a/\gamma^s)^r = 1$; this would imply that $a/\gamma^s = \lambda^i$, leading to $a \in Z$, contrary to $a \notin Z$. Hence, r does not divide j; similarly r does not divide k. Let $a_1 = a^k$ and $b_1 = b^j$; a direct computation from $ba = \lambda ab$ leads to $a_1b_1 = \mu b_1a_1$ where $\mu = \lambda^{-jk} \in Z$. Since the prime number r which is the order of λ does not divide j or k, $\lambda^{jk} \neq 1$ whence $\mu \neq 1$. Note that $\mu^r = 1$.

Let us see where we are. We have produced two elements a_1, b_1 such that:
(1) $a_1^r = b_1^r = \alpha \in Z$.
(2) $a_1b_1 = \mu b_1a_1$ with $\mu \neq 1$ in Z.
(3) $\mu^r = 1$.

We compute $(a_1^{-1}b_1)^r$; $(a_1^{-1}b_1)^2 = a_1^{-1}b_1a_1^{-1}b_1 = a_1^{-1}(b_1a_1^{-1})b_1 = a_1^{-1}(\mu a_1^{-1}b_1)b_1 = \mu a_1^{-2}b_1^2$. If we compute $(a_1^{-1}b_1)^3$ we find it equal to $\mu^{1+2}a_1^{-3}b_1^3$. Continuing we obtain $(a_1^{-1}b_1)^r = \mu^{1+2+\ldots+(r-1)}a_1^{-r}b_1^r = \mu^{1+2+\ldots+(r-1)} = \mu^{r(r-1)/2}$. If r is an odd prime, since $\mu^r = 1$, we get $\mu^{r(r-1)/2} = 1$, whence $(a_1^{-1}b_1)^r = 1$. Being a solution of $y^r = 1$, $a_1^{-1}b_1 = \lambda^i$ so that $b_1 = \lambda^i a_1$; but then $\mu b_1a_1 = a_1b_1 = b_1a_1$, contradicting $\mu \neq 1$. Thus if r is an odd prime number, the theorem is proved.

We must now rule out the case $r=2$. In that special situation we have two elements $a_1, b_1 \in D$ such that $a_1^2 = b_1^2 = \alpha \in Z$, $a_1b_1 = \mu b_1a_1$ where $\mu^2 = 1$ and $\mu \neq 1$. Thus $\mu = -1$ and $a_1b_1 = -b_1a_1 \neq b_1a_1$; in consequence, the characteristic of D is not 2. By Lemma 7.7 we can find elements $\zeta, \eta \in Z$ such that $1 + \zeta^2 - \alpha\eta^2 = 0$. Consider $(a_1 + \zeta b_1 + \eta a_1b_1)^2$; on computing this out we find that $(a_1 + \zeta b_1 + \eta a_1b_1)^2 = \alpha(1 + \zeta^2 - \alpha\eta^2) = 0$. Being in a division ring this yields that $a_1 + \zeta b_1 + \eta a_1b_1 = 0$; thus $0 \neq 2a_1^2 = a_1(a_1 + \zeta b_1 + \eta a_1b_1) + (a_1 + \zeta b_1 + \eta a_1b_1)a_1 = 0$. This contradiction finishes the proof and Wedderburn's theorem is established.

(*end of typical modern proof*)

Before we start to feel pleased with ourselves at the improvement, let me point out that there are two kinds of calculation in this text. One kind occurs in formulas, such as

$$\lambda^r a^r=(bab^{-1})^r=ba^rb^{-1}=a^r$$

$$b^r=\lambda^r b^r=(\lambda b)^r=(a^{-1}ba)^r=a^{-1}b^r a$$

$$(a_1^{-1}b_1)^2=a_1^{-1}b_1a_1^{-1}b_1=a_1^{-1}(b_1a_1^{-1})b_1=a_1^{-1}(\mu a_1^{-1}b_1)b_1=\mu a_1^{-2}b_1^2$$

$$(a_1^{-1}b_1)^r=\mu^{1+2+\ldots+(r-1)}a_1^{-r}b_1^r=\mu^{1+2+\ldots+(r-1)}=\mu^{r(r-1)/2}$$

This kind uses algebra well. The other kind occurs in the English text between the formulas. A proof is a boolean calculation, and in the current state of mathematics, as in the example, it is usually conducted mostly in natural language. Words like “consequently”, “implying”, “there is/are”, “however”, “thus”, “hence”, “since”, “forces”, “if...then”, “in consequence of which”, “from which we get”, “whence”, “would imply”, “contrary to”, “so that”, “contradicting” suggest boolean operators; all the bookkeeping sentences suggest the structure of a boolean expression. A formal proof is a boolean calculation using boolean algebra; when we learn to use it well, it will enable us to go farther, faster, more succinctly, and with much greater certainty. But there is a great resistance in the mathematical community to formal proof, especially from those who are most expert at informal proof. They complain that formal proof loses meaning, replacing it with symbol manipulation. The current state of boolean algebra, not as an object of study but as a tool for use, is very much the same as number algebra was 5 centuries ago.

Traditional Terminology

Formal logic has developed a traditional terminology that its students are required to learn. There are terms which are said to have values. There are formulas, also known as propositions or sentences, which are said not to have values, but instead to be true or false. Operators (+, –) join terms, while connectives ($\wedge$, $\vee$) join formulas. Some terms are boolean, and they have the value *true* or *false* , but that's different from being true or false. It is difficult to find a definition of predicate, but it seems that a boolean term like $x=y$ stops being a boolean term and mysteriously starts being a predicate when we admit the possibility of using quantifiers ($\exists$, $\forall$). Does $x+y$ stop being a number term if we admit the possibility of using summation and product (Σ, Π)? There are at least three different equal signs: $=$ for terms, and then $\Leftrightarrow$ and $\equiv$ for formulas and predicates, with one of them carrying an implicit universal quantification. We can even find a peculiar mixture in some textbooks, such as the following:

$$a+b = a \vee a+b = b$$

Here, a and b are boolean variables, $+$ is a boolean operator (disjunction), $a+b$ is a boolean term (having value *true* or *false*), $a+b = a$ and $a+b = b$ are formulas (so they are true or false), and finally $\vee$ is a logical connective.

Fortunately, in the past few decades there has been a noticeable shift toward erasing the distinction between being true or false and having the value *true* or *false*. It is a shift toward the calculational style of proof. But we have a long way to go yet, as I find whenever I ask my beginning students to prove something. If I ask them to prove something of the form $a \equiv b$ they happily try to do so. If I ask them to prove something of the form $a \vee b$, they take an unwittingly constructivist interpretation, and suppose I want them to prove a or prove b; they cannot understand the phrase "prove $a \vee b$" otherwise, because "or" isn't a verb! Here is an even more blatant example: prove $a \oplus b$ where $\oplus$ is pronounced "exclusive or". They cannot even start because they need something that looks grammatically like a proposition or sentence. If I change it to either $(a \oplus b) \equiv true$ or to $a \not\equiv b$ they are happy. The same lack of understanding can be found in many introductory programming texts where boolean expressions are not taught in their generality but as comparisons because comparisons have verbs!

while *flag=true* **do** *something*

Our dependence on natural language for the understanding of boolean expressions is a serious impediment.

Traditional Notations

Arithmetic notations are reasonably standard throughout the world. The expression

738 + 45 = 783

is recognized and understood by schoolchildren almost everywhere. But there are no standard boolean notations. Even the two boolean constants have no standard symbols. Symbols in use include

true	t	tt	T	1	0	1=1
false	f	ff	F	0	1	1=2

Quite often the boolean constants are written as 1 and 0, with + for disjunction, juxtaposition for conjunction, and perhaps – for negation. With this notation, here are some laws.

$x(y+z) = xy + xz$

$x + yz = (x+y)(x + z)$

$x + -x = 1$

$x(-x) = 0$

The overwhelming reaction of algebraists is: it doesn't matter which symbols are used. Just introduce them, and get on with it. But to apply an algebra, one must recognize the patterns, matching laws to the expression at hand. The laws have to be familiar. The first law above coincides with number algebra, but the next three clash with number algebra. It takes an extra moment to think which algebra I am using as I apply a law. The logician

R.L. Goodstein [3] chose to use 0 and 1 the other way around, which slows me down a little more. A big change, like using + as a variable and *x* as an operator, would slow me down a lot. I think it matters even to algebraists because they too have to recognize patterns. To a larger public, the reuse of arithmetic symbols with different meanings is an insurmountable obstacle. And when we mix arithmetic and boolean operators in one expression, as we often do, it is impossible to disambiguate.

The most common notations for the two boolean constants found in programming languages and in programming textbooks seem to be *true* and *false* . I have two objections to these symbols. The first is that they are clumsy. Number algebra could never have advanced to its present state if we had to write out words for numbers.

seven three eight + four five = seven eight three

is just too clumsy, and so is

true ∧ *false* ∨ *true* ≡ *true*

Clumsiness may seem minor, but it can be the difference between success and failure in a calculus.

My second, and more serious, objection is that the words *true* and *false* confuse the algebra with an application. One of the primary applications of boolean algebra is to formalize reasoning, to determine the truth or falsity of some statements from the truth or falsity of others. In that application, we use one of the boolean values to represent an arbitrary true statement, and the other to represent an arbitrary false statement. So for that application, it seems reasonable to call them *true* and *false* . The algebra arose from that application, and it is so much identified with it that many people cannot separate them. But of course boolean expressions are useful for describing anything that comes in two kinds. We apply boolean algebra to circuits in which there are two voltages. We sometimes say that there are 0s and 1s in a computer's memory, or that there are *true*s and *false*s. Of course that's nonsense; there are neither 0s and 1s nor *true*s and *false*s in there; there are low and high voltages. We need symbols that can represent truth values and voltages equally well.

Boolean expressions have other applications, and the notations we choose should be equally appropriate for all of them. Computer programs are written to make computers work in some desired way. Before writing a program, a programmer should know which ways are desirable and which are not. That divides computer behavior into two kinds, and we can use boolean expressions to represent them. A boolean expression used this way is called a specification. We can specify anything, not just computer behavior, using boolean expressions. For example, if you would like to buy a table, then tables are of two kinds:

those you find desirable and are willing to buy, and those you find undesirable and are not willing to buy. So you can use a boolean expression as a table specification. Acceptable and unacceptable human behavior is specified by laws, and boolean expressions have been proposed as a better way than legal language for writing laws.

For symbols that are independent of the application, I propose the lattice symbols $\top$ and $\bot$, pronounced "top" and "bottom". Since boolean algebra is the mother of all lattices, I think it is appropriate, not a misuse of those symbols. They can equally well be used for true and false statements, for high and low voltages (power and ground), for satisfactory and unsatisfactory tables, for innocent and guilty behavior, or any other opposites.

We seem to be settling on the symbols $\wedge$ and $\vee$ for conjunction and disjunction, although they are still not universal. They are explained by the use of the words "and" and "or"; even when they are explained by their "truth tables" we remember them by the fact that $x \wedge y$ is $\top$ exactly when both x and y are $\top$, and similarly for $\vee$.

We are less settled on a symbol for implication. Symbols in use include

$\rightarrow \quad \Rightarrow \quad \therefore \quad \supset$

The usual explanation says it means "if then", followed by a discussion about the meaning of "if then". Apparently, people find it difficult to understand an implication whose antecedent is *false*. Such an implication is called "counter-factual". For example, Charles Navarre declared [4]: "If my mother had been a man, I'd be the king of France.". Some people are uneasy with the idea that *false* implies anything, so some researchers in Artificial Intelligence have proposed a new definition of implication. The following truth table shows both the old and new definitions.

a	b	old $a \Rightarrow b$	new $a \Rightarrow b$
true	*true*	*true*	*true*
true	*false*	*false*	*false*
false	*true*	*true*	*unknown*
false	*false*	*true*	*unknown*

where *unknown* is a third boolean value. When the antecedent is *false*, the result of the new kind of implication is *unknown*. This is argued to be more intuitive. I believe this proposal betrays a serious misunderstanding of the use of logic. When someone makes a statement, they are saying that the statement is *true*. Even if the statement is "if *a* then *b* " and *a* is known to be *false*, nonetheless we are being told that "if *a* then *b* " is *true*. It is the consequent *b* that is unknown. And that is represented perfectly by the old implication: there are two rows in which *a* is *false* and $a \Rightarrow b$ is *true*; on one of these rows, *b* is *true*, and on the other *b* is *false*.

There are two other symbols

$\longrightarrow$ $\vdash$

that mean something like implication. We are told that these are not implication, but you must admit that the distinction is subtle. The explanations sound similar: if the left (or top) side is a theorem, then the right (or bottom) side is too. And the Deduction Theorem says that $\vdash$ coincides with implication for a large part of logic. It is just such complications that keep logic out of use, even by mathematicians.

In case you think that confusion is just for beginners or philosophers, consider the explanation of implication in *Contemporary Logic Design*, 1994 [5]:

> "As an example, let's look at the following logic statement:
>
> IF the garage door is open
>
> AND the car is running
>
> THEN the car can be backed out of the garage
>
> It states that the conditions — the garage is open and the car is running — must be true before the car can be backed out. If either or both are false, then the car cannot be backed out. If we determine that the conditions are valid, then mathematical logic allows us to infer that the conclusion is valid."

Even a Berkeley electrical engineering professor can't get implication right.

Implication is best presented as an ordering, and for primary school students, all the explanation necessary can be carried by its name. If we are still calling the boolean values "true" and "false" then we can call it "falser than or equal to", or if you prefer, "as false as". It is easy to see that *false* is falser than or equal to *true* , and that *false* is falser than or equal to *false* . As we get into boolean expressions that use other types, this explanation remains good: $x>6$ is falser than or equal to $x>3$, as a sampling of evaluations illustrates. If we are calling the boolean values "top" and "bottom", we can say "lower than or equal to" for implication. With this new pronunciation and explanation, three other neglected boolean operators become familiar and usable; they are "higher than or equal to", "lower than", and "higher than". For lack of a name and symbol, the last two operators have been treated like shameful secrets, and shunned. Even implication has often been defined as a "secondary" operator in terms of the "primary" operators negation and disjunction:

$$(a \Longrightarrow b) \equiv \neg a \vee b$$

This avoids the philosophical explanation, but it makes an unsupportable distinction between "primary" and "secondary" operators, and hides the fact that it is an ordering. If we present implication as an ordering, as I prefer, then we face the problem of how to use this ordering in the formalization of natural language reasoning. Philosophers and linguists can help, or indeed dominate in this difficult and important area. But we shouldn't let the

complexities of this application of boolean algebra complicate the algebra, any more than we let the complexities of the banking industry complicate the definition of arithmetic.

That implication is the boolean ordering, with $\top$ and $\bot$ at the extremes, is not known to all who use boolean algebra. In the specification language Z, boolean expressions are used as specifications. Specification A refines specification B if all behavior satisfying A also satisfies B. Although increasing satisfaction is exactly the implication ordering, the designers of Z defined a different, complicated ordering for refinement where $\top$ is <u>not</u> satisfied by all computations, only by terminating computations, and $\bot$ <u>is</u> satisfied by some computations, namely nonterminating computations. When even they can get it wrong, logic is not well understood or used.

Symmetry and Duality

In choosing binary infix symbols, there is a simple principle that really helps our ability to calculate: we should choose symmetric symbols for symmetric operators, and asymmetric symbols for asymmetric operators, and choose the reverse of an asymmetric symbol for the reverse operator. The benefit is that we get a lot of laws for free: we can write an expression backwards and get an equivalent expression. For example, $x + y < z$ is equivalent to $z > y + x$. By this principle, the arithmetic symbols $+ \times < > =$ are well chosen but $-$ and $\neq$ are not. The boolean symbols $\wedge \vee \Rightarrow \Leftarrow \equiv \oplus$ are well chosen, but $\not\equiv$ is not.

Duality can be put to use, just like symmetry, if we use vertically symmetric symbols for self-dual operators, and vertically asymmetric operators for non-self-dual operators with the vertical reverse for their duals. The laws we get for free are: to negate an expression, turn it upside down. For example, $(\top \wedge -\bot) \vee \bot$ is the negation of $(\bot \vee -\top) \wedge \top$ if you allow me to use the vertically symmetric symbol $-$ for negation, which is self-dual. There are two points that require attention when using this rule. One is that parentheses may need to be added to maintain the precedence; but if we give dual operators the same precedence, there's no problem. The other point is that variables cannot be flipped, so we negate them instead (since flipping is equivalent to negation). The well-known example is deMorgan's law: to negate $a \vee b$, turn it upside down and negate the variables to get $-a \wedge -b$. By this principle, the symbols $\top \bot - \wedge \vee$ are well chosen, but $\Rightarrow \Leftarrow \equiv \not\equiv \oplus$ are not. By choosing better symbols we can let the symbols do some of the work of calculation, moving it to the level of visual processing.

Booleans and Numbers

I have long thought it was a mistake to identify booleans with numbers, even if just by the reuse of symbols. It's a type error. The C language continues the mistake. Thus we can write (1 && 1) + 1 and get 2. I have recently changed my mind. I now think the association —even the identification— between booleans and numbers is right, but not the association we are used to.

I like to prove things about the execution time of programs, and for that purpose I use a number system extended with an infinity (because that's the execution time of some programs). For some purposes I use integers extended with infinity, and for others I use reals extended with infinity. For a list of axioms of this arithmetic, please see the appendix; for more detail, please see [6]. Here I mention only that infinity is maximum $x \leqslant \infty$ and it absorbs additions $\infty+1=\infty$. For my purposes, I have not needed a lot of infinite cardinalities; a single infinity is enough. The association I want to make between booleans and numbers is the following.

	boolean	number	
top	$\top$	∞	infinity
bottom	$\bot$	$-\infty$	minus infinity
negation	$\neg$	$-$	negation
conjunction	$\wedge$	$\downarrow$	minimum
disjunction	$\vee$	$\uparrow$	maximum
“nand”	$\Uparrow$	$\twoheaddownarrow$	negation of minimum
“nor”	$\Downarrow$	$\twoheaduparrow$	negation of maximum
implication	$\Rightarrow$	$\leqslant$	order
reverse implication	$\Leftarrow$	$\geqslant$	reverse order
strict implication	$\rhd$	$<$	strict order
strict reverse implication	$\lhd$	$>$	strict reverse order
equivalence	$\equiv$	$=$	equality
exclusive or	$\oplus$	$\neq$	inequality

I have temporarily invented a few symbols to fill in some gaps. The remaining three unary operators and six binary operators are degenerate, so I have not included them. With this association, all number laws employing only these operators correspond to boolean laws. For example,

boolean law	number law
$\top \equiv \neg\bot$	$\infty = - -\infty$
$a \equiv \neg\neg a$	$x = - -x$
$a \vee \top \equiv \top$	$x \uparrow \infty = \infty$
$a \wedge \bot \equiv \bot$	$x \downarrow -\infty = -\infty$
$a \vee \bot \equiv a$	$x \uparrow -\infty = x$
$a \wedge \top \equiv a$	$x \downarrow \infty = x$
$a \Rightarrow \top$	$x \leqslant \infty$
$\bot \Rightarrow a$	$-\infty \leqslant x$
$a \vee (b \wedge c) \equiv (a \vee b) \wedge (a \vee c)$	$x \uparrow (y \downarrow z) = (x \uparrow y) \downarrow (x \uparrow z)$
$a \wedge (b \vee c) \equiv (a \wedge b) \vee (a \wedge c)$	$x \downarrow (y \uparrow z) = (x \downarrow y) \uparrow (x \downarrow z)$
$a \vee b \equiv \neg(\neg a \wedge \neg b)$	$x \uparrow y = -(-x \downarrow -y)$
$a \wedge b \equiv \neg(\neg a \vee \neg b)$	$x \downarrow y = -(-x \uparrow -y)$

There are, however, boolean laws that do not correspond to number laws. For example,

boolean law	number non-law
$\top$	∞
$\neg\bot$	$- -\infty$
$a \vee \neg a \equiv \top$	$x \uparrow -x = \infty$
$a \wedge \neg a \equiv \bot$	$x \downarrow -x = -\infty$
$(\top \Rightarrow a) \equiv a$	$(\infty \leqslant x) = x$
$(\top \equiv a) \equiv a$	$(\infty = x) = x$

Number algebra has developed by the desire to solve equations, or more generally, to solve boolean expressions. This has resulted in an increasing sequence of domains, from naturals to integers to rationals to reals to complex numbers. As we gain solutions, we lose laws.

small domain	$\longleftrightarrow$	large domain
more laws	$\longleftrightarrow$	fewer laws
fewer solutions	$\longleftrightarrow$	more solutions

This is because a law is essentially a universal quantification, and a boolean expression to be solved is essentially an existential quantification.

law: $\forall$*variables*: *domains*$\cdot$ *boolean expression*

solution: $\exists$*variables*: *domains*$\cdot$ *boolean expression*

As the domain of an operation or function grows, we do not change its symbol; addition is still denoted + as we go from naturals to complex numbers. I will not argue whether the naturals are a subset of the complex numbers or just isomorphic to a subset; for me the question has no meaning. But I do argue that it is important to use the same notation for

natural 1 and complex 1 because they behave the same way, and for natural + and complex + because they behave the same way on their common domain. To be more precise, all laws of complex arithmetic that can be interpreted over the naturals are laws of natural arithmetic, and all equations (or more generally, boolean expressions) over the naturals retain the same solutions over the complex numbers. The reason we must use the same symbols is so that we do not have to relearn all the laws and solutions as we enlarge or shrink the domain.

I have been hammering on a point that I expect is not contentious. If I have your agreement, then you must conclude, as I must, that the symbols of boolean algebra and arithmetic must be unified. The question whether boolean is a different type from number is no more relevant than the question whether natural and integer are different types. What's important is that laws and solutions are learned once, in a unified system, not twice in conflicting systems. And that matters both to professional mathematicians who must apply laws and solve, and to primary school students who must struggle to learn what will be useful to them.

Unified Algebra

Here is my proposal for the symbols of a unified algebra.

	unified	
top	$\top$	infinity
bottom	$\bot$	minus infinity
negation	$-$	negation
conjunction	$\wedge$	minimum
disjunction	$\vee$	maximum
“nand”	$\triangle$	negation of minimum
“nor”	$\triangledown$	negation of maximum
implication	$\leqslant$	order
reverse implication	$\geqslant$	reverse order
strict implication	$<$	strict order
strict reverse implication	$>$	strict reverse order
equivalence	$=$	equality
exclusive or	$\neq$	inequality

The symbols $-$ $\leqslant$ $\geqslant$ $<$ $>$ $=$ are world-wide standards, used by school children in all countries, so I dare not suggest any change to them. The symbol $\neq$ for inequality is the

next best known, but I have dared to stand up the slash so that all symmetric operators have symmetric symbols and all asymmetric operators have asymmetric symbols. (Although it was not a consideration, $\neq$ also looks more like $\oplus$.) There are no standard symbols for minimum and maximum, so I have used the boolean conjunction and disjunction symbols. The "nand" symbol is a combination of the "not" and "and" symbols, and similarly for "nor". Duality has been sacrificed to standards; the pair $\leqslant$ $<$ are duals, so they ought to be vertical reflections of each other; similarly the pair $\geqslant$ $>$, and also $=$ $\neq$. Since we now have a unified boolean and number algebra, I might mention that addition and subtraction are self-dual, and happily $+$ and $-$ are vertically symmetric; multiplication is not self-dual, but $\times$ is unfortunately vertically symmetric.

Having unified the symbols, I suppose we should also unify the terminology. I vote for the number terminology in the right column, except that I prefer to call $\top$ and $\bot$ "top" and "bottom".

In the unified algebra, the fact that $x=-x$ has no boolean solution but does have an integer solution is no more bothersome than that $x^2=2$ has no integer solution but does have a real solution. The fact that $x\neq -x$ is a boolean law but not an integer law is no more bothersome than that $x^2\neq 2$ is an integer law but not a real law.

Quantifiers

I am told that the symbols $\forall$ and $\exists$ send engineers running, and I don't blame them. For me, the problem with these symbols is that they are associated with the words "all" and "exist". I am truly sorry the word "existence" was ever used in mathematics. We can certainly apply mathematics to problems concerning the existence of something in the application area, and then I once again leave it to philosophers or linguists to determine how best to apply it, and how well the mathematical expressions can represent the existence of some application objects. But I don't want any debate about existence within mathematics; to me, mathematical existence is meaningless.

The nicest, simplest presentation of quantifiers, perhaps due to Curry, begins with the treatment of functions due to Church. I write a function, or local scope, according to the following example:

$\langle n\colon nat\cdot\ n+1\rangle$

Originally, instead of angle brackets, Church used a long hat over the expression to denote the scope of the variable. Due to the obvious typesetting difficulty, Church was persuaded to write the hat in front of the expression rather than over it, and the most similar available

character was λ; thus the lambda calculus was born. Following van de Snepscheut, I have returned to the original spirit, and use angle brackets to delimit scope. Next, I want to get rid of the idea that all possible variables (infinitely many of them) already "exist", and the function notation (λ or $\langle\ \rangle$) is said to "bind" variables, and any variable that is not bound remains "free". I prefer the programmer's terminology of "local" and "global" variables. Variables do not automatically "exist"; they are introduced (rather than bound) by the function notation.

A local variable can be instantiated, in other words a function can be applied to an argument, but at the moment I am interested in applying operators to functions. If the body of a function is a number expression, then we can apply $+$ to obtain the sum of the function results. For example,

$$+\langle n: nat\cdot\ 1/2^n\rangle$$

There is no syntactic ambiguity caused by this use of $+$, so no need to employ another symbol Σ for addition. The introduction of the dummy variable and its domain are exactly the job of the function notation, so no need to employ another notation for variable introduction with quantifiers. And the notation generalizes to other binary associative symmetric operators, such as

$$\times\langle n: nat\cdot\ 1/2^n\rangle$$
$$\wedge\langle n: nat\cdot\ n>5\rangle$$
$$\vee\langle n: nat\cdot\ n>5\rangle$$

There are no scary symbols. We talk about a maximum, not existence, because it <u>is</u> a maximum, not existence. By applying $=$ and $\neq$ to functions we obtain the two independent parity quantifiers. Even set formation, limits, and integrals can be treated this way.

The sum of two rationals is rational; the sum of infinitely many rationals may not be rational. Nonetheless, we continue to use the word "sum" and symbol $+$. Similarly, I see no need to switch teminology from "maximum" to "least upper bound" as we generalize $\vee$ from two operands to infinitely many; we just have to learn that the maximum of a set may not be in the set.

If function f has domain D, then $f\ =\ \langle x: D\cdot\ fx\rangle$, so quantifications traditionally written

$$\Sigma x: D\cdot\ fx \qquad\qquad \forall x: D\cdot\ Px$$

which we can now write as

$$+\langle x: D\cdot\ fx\rangle \qquad\qquad \wedge\langle x: D\cdot\ Px\rangle$$

can be written even more succinctly as

$$+f \qquad\qquad \wedge P$$

Using juxtaposition for composition, deMorgan's laws

$\neg(\forall x{:}\, D{\cdot}\ Px) \ \equiv\ (\exists x{:}\, D{\cdot}\ \neg Px)$ $\qquad$ $\neg(\exists x{:}\, D{\cdot}\ Px) \ \equiv\ (\forall x{:}\, D{\cdot}\ \neg Px)$

become

$-\!\wedge P \ =\ \vee\!-P$ $\qquad$ $-\!\vee P \ =\ \wedge\!-P$

or even more succinctly

$(-\!\wedge) \ =\ (\vee\!-)$ $\qquad$ $(-\!\vee) \ =\ (\wedge\!-)$

The Specialization and Generalization laws say that if $y{:}\, D$,

$(\forall x{:}\, D{\cdot}\ Px) \ \Rightarrow\ Py$ $\qquad$ $Py \ \Rightarrow\ (\exists x{:}\, D{\cdot}\ Px)$

They now become

$\wedge P \ \leqslant\ Py$ $\qquad$ $Py \ \leqslant\ \vee P$

which say that the minimum item is less than or equal to any item, and any item is less than or equal to the maximum item. These laws hold for all numbers, not just for the boolean subset.

To define quantifiers formally, we have to say, for each domain constructor, how they apply to functions with such domains. The axioms follow a pattern:

$\wedge\langle x{:}\, \{\}{\cdot}\ e\rangle \ =\ \top$

$\wedge\langle x{:}\, \{y\}{\cdot}\ e\rangle \ =\ \langle x{:}\, \{y\}{\cdot}\ e\rangle\, y$

$\wedge\langle x{:}\, A\cup B{\cdot}\ e\rangle \ =\ \wedge\langle x{:}\, A{\cdot}\ e\rangle \wedge \wedge\langle x{:}\, B{\cdot}\ e\rangle$

$\vee\langle x{:}\, \{\}{\cdot}\ e\rangle \ =\ \bot$

$\vee\langle x{:}\, \{y\}{\cdot}\ e\rangle \ =\ \langle x{:}\, \{y\}{\cdot}\ e\rangle\, y$

$\vee\langle x{:}\, A\cup B{\cdot}\ e\rangle \ =\ \vee\langle x{:}\, A{\cdot}\ e\rangle \vee \vee\langle x{:}\, B{\cdot}\ e\rangle$

$+\langle x{:}\, \{\}{\cdot}\ e\rangle \ =\ 0$

$+\langle x{:}\, \{y\}{\cdot}\ e\rangle \ =\ \langle x{:}\, \{y\}{\cdot}\ e\rangle\, y$

$+\langle x{:}\, A\cup B{\cdot}\ e\rangle + +\langle x{:}\, A\cap B{\cdot}\ e\rangle \ =\ +\langle v{:}\, A{\cdot}\ e\rangle + +\langle v{:}\, B{\cdot}\ e\rangle$

$\times\langle x{:}\, \{\}{\cdot}\ e\rangle \ =\ 1$

$\times\langle x{:}\, \{y\}{\cdot}\ e\rangle \ =\ \langle x{:}\, \{y\}{\cdot}\ e\rangle\, y$

$\times\langle x{:}\, A\cup B{\cdot}\ e) \times \times\langle x{:}\, A\cap B{\cdot}\ e\rangle \ =\ \times\langle x{:}\, A{\cdot}\ e\rangle \times \times\langle x{:}\, B{\cdot}\ e\rangle$

If there are other domain constructors, there are other axioms. A domain can even be defined by saying how a quantifier applies to functions with that domain. For example, *nat* can be defined by

$\wedge\langle x{:}\, nat{\cdot}\ Px\rangle \ =\ P0 \wedge \wedge\langle x{:}\, nat{\cdot}\ Px \leqslant P(x{+}1)\rangle$

or dually (renaming P as its negation)

$\vee\langle x{:}\, nat{\cdot}\ Px\rangle \ =\ P0 \vee \vee\langle x{:}\, nat{\cdot}\ Px < P(x{+}1)\rangle$

Those who dislike formal definitions may have a desire to say in natural language how $\wedge$ applies to all boolean functions, regardless of how the domain was constructed. They may want to say that the result is $\top$ exactly when <u>all</u> range elements are $\top$. The word "all" sounds clear and unambiguous, but we have enough experience to know that it is far from clear and unambiguous. (Are so-called "undefined" range elements included?) Natural language definitions lead to a lot of arguments, and I have lost patience with them. Only a formal definition, equivalent to an automated theorem prover, is clear and unambiguous.

Here's an interesting experiment: ask a colleague if $(\forall x \cdot Px) \Rightarrow (\exists y \cdot Qy)$ is equivalent to $\exists x \cdot \exists y \cdot (Px \Rightarrow Qy)$ and then listen to their efforts to find the answer. They probably don't find it obvious. Those who reason informally say things like "suppose all x have property P", and "suppose some y has property Q". They are led into case analyses by treating $\forall$ and $\exists$ as abbreviations for "for all" and "there exists" (as they originally were). Of the very few who reason formally, most don't know many laws; perhaps they start by getting rid of the implications in favor of $\neg$ and $\vee$, then use deMorgan's laws. Let me rewrite the question in the new notations.

$$(\wedge P \leq \vee Q) \;=\; \vee\langle x \cdot \vee\langle y \cdot Px \leq Qy\rangle\rangle$$

On the left, it says the minimum P is at most the maximum Q. On the right it says that some P is at most some Q. Now it's more obviously a theorem, not just for booleans but for all numbers. To prove it, one should know (or prove) laws like

$$(\wedge P \leq q) \;=\; \vee\langle x \cdot Px \leq q\rangle$$

(the minimum P is less than or equal to q if and only if some P is less than or equal to q), and dually

$$(p \leq \vee Q) \;=\; \vee\langle y \cdot p \leq Qy\rangle$$

(p is less than or equal to the maximum Q if and only if p is less than or equal to some Q). The proof is then

$$\begin{aligned} & \vee\langle x \cdot \vee\langle y \cdot Px \leq Qy\rangle\rangle \\ = \quad & \vee\langle x \cdot Px \leq \vee Q\rangle \\ = \quad & (\wedge P \leq \vee Q) \end{aligned}$$

It is not the presence of quantifiers that moves us up from zero-order logic to first-order logic, but the presence of functions, with domains restricted to zero-order expressions. With unrestricted domains, we move up again to higher-order logic. Logicians seem to like to settle the question "which logic are we in" before they do any reasoning. Can you imagine asking a working mathematician or engineer to decide whether they will be using functions, and if so, what will be their domains, before beginning their work? The answer would be: I'll use whatever I need when I need it.

Metalogic

Almost always, number algebra is presented without a metanotation, while logic is presented with one. The distinction between the metanotation and the object notation is not easily appreciated by students, or by many teachers.

Logicians study logic. There are no applied logicians who use logic to study something else. In the study of logic, at or near the beginning, logicians present the very important symbol $\vdash$ to represent theoremhood. I ask you to put yourself in the place of a beginning student. This symbol is applied to a boolean expression just like the boolean operators; but we know all the boolean operators and this isn't one of them. To say that it is a "meta-operator" just labels it, and doesn't explain it. Saying that it applies to the form, rather than the meaning, is confusing too, since the entire point of the algebra is to enable us to work with the form and ignore the meaning. In my opinion, the use of meta-level operators is unnecessary and ill-conceived.

To apply an operator to the form of an expression, we do not need any new kind of operator. Rather, we need to do exactly what Gödel did when he encoded expressions, but we can use a better encoding. We need to do exactly what programmers do: distinguish program from data. One person's program may be a compiler writer's data, but when it is data, it is a character string. We should apply $\vdash$ to character strings. The character string "$a \vee -a$" can be used as a code for the expression $a \vee -a$. We define $\vdash s$ according to the structure of boolean expressions so that $\vdash s$ is a theorem when the boolean expression represented by string s is a theorem. We could also define another operator $\dashv$ that serves a dual role to $\vdash$: it applies to character strings so that $\dashv s$ is an antitheorem when the boolean expression represented by string s is an antitheorem. By "antitheorem" I mean those boolean expressions that can be simplified (proven equal) to $\bot$. In some logics, those having negation and an appropriate proof rule, "antitheorem" means "negation of a theorem", but not in all. It deserves a name and symbol just as much as $\bot$ does. It's surprising that the dual of theorem has not been invented before.

I propose that logicians can improve metalogic in another way, by taking another lesson from programming. Instead of $\vdash$ and $\dashv$, we need only one operator to serve both purposes. It is called an interpreter. I want $\mathbb{I} s$ to be a theorem if and only if s represents a theorem, and an antitheorem if and only if s represents an antitheorem. It is related to $\vdash$ and $\dashv$ by the two implications

$$\vdash s \leqslant \mathbb{I} s \leqslant -\dashv s$$

In fact, if we have defined $\vdash$ and $\dashv$, those implications define $\mathbb{I}$. But I want $\mathbb{I}$ to

replace ⊢ and ⊣ so I shall instead define it by showing how it applies to every form of boolean expression. Here is the beginning of its definition.

$$\mathbb{I}\text{“}\top\text{”} = \top$$
$$\mathbb{I}\text{“}\bot\text{”} = \bot$$
$$\mathbb{I}(\text{“}-\text{”}\ s) = -\,\mathbb{I}\,s$$
$$\mathbb{I}(s\ \text{“}\wedge\text{”}\ t) = \mathbb{I}\,s \wedge \mathbb{I}\,t$$
$$\mathbb{I}(s\ \text{“}\vee\text{”}\ t) = \mathbb{I}\,s \vee \mathbb{I}\,t$$

And so on. In a vague sense $\mathbb{I}$ acts as the inverse of quotation marks; it “unquotes” its operand. That is what an interpreter does: it turns passive data into active program. It is a familiar fact to programmers that we can write an interpreter for a language in that same language, and that is just what we are doing here. Interpreting (unquoting) is exactly what logicians call Tarskian semantics. In summary, an interpreter is a better version of ⊢, and strings make meta-level operators unnecessary.

Proof Rules

You cannot learn a programming language by reading an interpreter for it written in that same language. And you cannot learn logic, or a logic, by reading an interpreter for it written in logic. Not only is it inscrutable to a novice, but also it may be subject to more than one interpretation. We can, of course, present one formalism with the aid of another, a metanotation. But my goal is to teach boolean algebra to a wide audience, and for that purpose I do not think it is profitable to require them to learn another formalism first. I think it should be presented as number algebra is presented, with a little natural language and a lot of axioms, because axioms don't use any extra notations.

Here are the proof rules I am using. The rules place boolean expressions into two classes: theorems and antitheorems. In an incomplete logic, some boolean expressions will remain unclassified. Note that the rules never mention any boolean operators.

Axiom Rule — If a boolean expression is an axiom, then it is a theorem. If a boolean expression is an antiaxiom, then it is an antitheorem.

Evaluation Rule — If all the boolean subexpressions of a boolean expression are classified, then it is classified according to the evaluation tables (truth tables).

Completion Rule — If a boolean expression contains unclassified boolean subexpressions, and all ways of classifying them place it in the same class, then it is in that class.

Consistency Rule — If a classified boolean expression contains boolean subexpressions, and exactly one way of classifying them is consistent, then they are classified that way.

Instance Rule — If a boolean expression is classified, then all its instances have that same classification.

There can be both axioms and antiaxioms; $\top$ is an axiom and $\bot$ is an antiaxiom. If the logic includes both negation and the Consistency Rule, we can dispense with the words "antiaxiom" and "antitheorem", but I suggest we keep them for the sake of duality. The boolean operators all enter together with equal status via the Evaluation Rule. The Completion Rule includes, as a special case, that $a \vee -a$ is a theorem; constructivists will omit this rule. Consistency means that no boolean expression is classified both as a theorem and as an antitheorem; the Consistency Rule includes modus ponens as a special case. The Instance Rule refers to expressions obtained by replacing variables with expressions. In addition to these rules, we need only axioms (and perhaps antiaxioms), and the usual substitution rules.

Terms of Honor

My final comment concerns mathematical terminology intended to honor mathematicians. In some parts of mathematics it is standard: Lie algebra, Stone algebra, Jordan decomposition, Cayley transform, Hilbert space, Banach space, Hausdorff space, Borel measure, Lebesgue integration, Fredholm index, and so on. It is well known that the person so honored is sometimes the wrong person; often it is only one of many who equally deserve to have their names attached to the idea. I suspect that sometimes the intention is not so much to honor a person as to use the person's prestige to lend respectability to a subject. Even when the intention is to honor, the effect is to obscure and make the mathematics forbidding and inaccessible. It may be argued that this is good, keeping the uninitiated from thinking they understand when they don't. I know what nand and nor are, but I forget which is the Scheffer stroke and which the Pierce arrow. To say that an operator is symmetric or commutative is much more descriptive and understandable than calling it Abelian. DeMorgan's laws would be better named duality laws. We who are used to the terms forget what a barrier they pose to beginners.

The term “boolean algebra” honors George Boole. It is popularly thought that the word “algebra” honors someone, but according to scholars, that's a myth; it comes from an arabic word meaning “the reintegration and reunion of broken parts”. In any case, the word is now standard, known by average people everywhere. I revere George Boole and I want to honor him. The greatest honor I can think of is to make the algebra that he created a well known and well used tool, and to do that we might have to remove his name from it, and give it a more descriptive and accessible name, like “binary algebra”.

Conclusions

Logic has been well studied and is now well understood, but it is not well used. Programmers learn that logic is a foundation of programming, but they don't often use it to program. Mathematicians study about logic, but they don't often use it in their proofs. Logic is a tool, like a knife. People have looked at it from every angle; they've described how it works at great length; now it's time to pick it up and use it. To use logic well, one must learn it early, and practice a lot. Fancy versions of logic, such as three-valued logic, temporal logic, and metalogic, can be left to university study, but there is a simple basic algebra that can be taught early and used widely.

Number algebra is used by scientists and engineers everywhere. It is used by economists and architects. It is taught first to 6-year olds, very concretely as addition and subtraction of numbers. Then variables and equations are introduced, and always the applications are emphasized. As a result of that early and long education, scientists and engineers and mathematicians are comfortable with it. Boolean algebra, or logic, can be equally useful if it is taught the same way. At present, it is not in a good state for presentation to a wide audience. We need to simplify the terminology, choose some good symbols, adopt the view that proof is calculation, detach it from its dominant application in which the boolean values represent true and false statements, free it from philosophy and explain it as algebra.

There is a small advantage to choosing uniquely boolean symbols: we can give them a precedence after the arithmetic operators, which reduces the need for parentheses. On the other hand, there is a large advantage to uniting boolean and number symbols in the way I have suggested: the laws and solutions are familiar and can be interpreted either as booleans or numbers. In addition, by placing booleans in the same context as numbers, we move quickly away from philosophical explanations, and we are less likely to introduce strange kinds of implication or strange kinds of logic. The fact that the booleans can be embedded in the extended integers just as smoothly as the integers are embedded in the rationals seems a compelling reason to do so.

Quantifiers can be simplified, made uniform, and generalized by treating them as operators on functions. We should stop speaking about “existence”, and speak instead about the maximum of a function. Similarly, we should stop speaking about “all”, and speak instead about the minimum of a function.

An interpreter serves the same purpose as the meta-level theoremhood operator with the added advantage that it gives antitheoremhood as well as theoremhood. And by applying it to strings, we don't need to introduce a separate meta-level of operators. Metalogic is an advanced topic, not a good introduction to logic for those who are new to the subject.

Appendix

Let d be a sequence of (zero or more) digits, let x, y, and z be any expressions. Then the following axioms are a unified boolean and number theory. The transitive operators $= \ < \ \leqslant$ are used in a continued (conjunctional) syntax. In addition to these axioms, we need proof rules (presented earlier), substitution rules, and evaluation tables (truth tables). Minimality is not claimed.

$\top$	
$-\bot$	
$x = x$	reflexivity
$(x=y) = (y=x)$	symmetry
$(x=y) \wedge (y=z) \ \leqslant \ (x=z)$	transitivity
$(x \neq y) = -(x=y)$	
$-(x<x)$	irreflexivity
$-((x<y) \wedge (y<x))$	antisymmetry
$(x<y) \wedge (y<z) \leqslant (x<z)$	transitivity
$-((x<y) \wedge (x=y))$	exclusivity
$(x \leqslant y) \ = \ (x<y) \vee (x=y)$	
$(x>y) \ = \ (y<x)$	
$(x \geqslant y) \ = \ (y \leqslant x)$	
$(x<y) \vee (x=y) \vee (x>y)$	totality, trichotomy
$d0+1 = d1$	counting
$d1+1 = d2$	counting
$d2+1 = d3$	counting
$d3+1 = d4$	counting
$d4+1 = d5$	counting
$d5+1 = d6$	counting

$d6+1 = d7$	counting
$d7+1 = d8$	counting
$d8+1 = d9$	counting
$d9+1 = (d+1)0$	counting
$+x = x$	identity
$x+0 = x$	identity
$x+y = y+x$	symmetry
$x+(y+z) = (x+y)+z$	associativity
$(\bot<x<\top) \leqslant ((x+y = x+z) = (y=z))$	cancellation
$--x = x$	self-inverse
$-(x+y) = -x + -y$	distributivity
$-(x\times y) = -x \times y$	semi-distributivity
$x-y = x + -y$	
$(\bot<x<\top) \leqslant (x-x = 0)$	inverse
$(\bot<y<\top) \leqslant (x - y + y = x)$	inverse
$(\bot<x<\top) \leqslant (x\times 0 = 0)$	base
$x\times 1 = x$	identity
$x\times y = y\times x$	symmetry
$x\times(y+z) = x\times y + x\times z$	distributivity
$x\times(y\times z) = (x\times y)\times z$	associativity
$(\bot<x<\top) \wedge (x\neq 0) \leqslant ((x\times y = x\times z) = (y=z))$	cancellation
$(\bot<y<\top) \wedge (y\neq 0) \leqslant (x/y\times y = x)$	inverse
$(\bot<x<\top) \leqslant (x^0 = 1)$	base
$x^1 = x$	identity
$x^{y+z} = x^y \times x^z$	
$x^{y\times z} = (x^y)^z$	
$\bot<0<1<\top$	direction
$(\bot<x<\top) \leqslant ((x+y < x+z) = (y<z))$	cancellation, translation
$(0<x<\top) \leqslant ((x\times y < x\times z) = (y<z))$	cancellation, scale
$(x<y) = (-y<-x)$	reflection
$\bot \leqslant x \leqslant \top$	extremes
$\top+1 = \top$	additive absorption
$(0<x) \leqslant (x\times\top = \top)$	multiplicative absorption
$(0<x) \leqslant (x/0 = \top)$	
$(\bot<x<\top) \leqslant (x/\top = 0)$	

Acknowledgment

Theo Norvell provided the Navarre quotation and the example from the Katz text.

References

[0] Unfortunately, 500 year old algebra texts are hard to find. This is not a quotation, but my own creation. I think it is representative of the work of the time.

[1] Robert Recorde: *the Whetstone of Witte*, London 1557, reprinted by Da Capo Press Amsterdam 1969

[2] I.N. Herstein: *Topics in Algebra*, Blaisdell 1964 p.323

[3] R.L. Goodstein: *Development of Mathematical Logic*, Springer-Verlag 1971

[4] quoted in Barbara W. Tuchman: *a Distant Mirrror: the Calamitous Fourteenth Century*, Knopf 1978

[5] Randy H. Katz: *Contemporary Logic Design*, Benjamin Cummings 1994 p.10

[6] E.C.R. Hehner: *a Practical Theory of Programming*, Springer-Verlag 1993