Unified Algebra

Eric C. R. Hehner
University of Toronto

Introduction

Mathematics has evolved. Bits of it are created by many people over time. The parts that survive are sometimes the best parts, and sometimes not. Sometimes the survival of a mathematical idea or notation has more to do with the personality of its creator than with its merit. Evolution tends to create complexity. Often there are a variety of notations that serve the same purpose (created by different people at different times) all in use in one paper. Occasionally it seems worthwhile to try to design mathematics, rather than just to evolve some more. When we design, we can strive for simplicity, which evolution never produces. When we design, we evaluate, keeping what is useful, unifying what is similar.

I present a unified algebra that includes what are commonly called boolean algebra, number algebra, sets, lists, functions, quantification, type theory, and limits. I present the notations and the rules for the conduct of algebra, but it is not the purpose here to explore the possibilities for their use. I am laying foundations, not building upon them; I am designing the instrument, not playing the music. To appreciate the instrument, which is the algebra I present, I rely on the reader's experience in using algebra. For motivations, justifications, and commentary, I refer the reader to [1]. The viewpoint I adopt throughout is formalist.

The algebra is presented from the very beginning, leaving out nothing. That makes parts of the presentation very basic, and for some readers, boring. But other readers may appreciate the care and effort required to design a simple and general algebra. I begin with boolean algebra, renamed “binary algebra”, and its two extremes are renamed “top” and “bottom”. That's the only new terminology. By contrast, the standard terminology that I won't be using includes: boolean, true, false, proposition, sentence, term, formula, conjunction, conjunct, disjunction, disjunct, implication, implies, antecedent, consequent, axiom, theorem, lemma, proof, inference, entailment, syntax, semantics, valid, predicate, quantifier, quantification, universal, existential, and existence. I consider symbols and terminology to be a cost, not a benefit, when defining mathematical structures. Unified algebra gives us much more mathematics for less cost than usual.

Algebra

We will soon introduce binary algebra, ternary algebra, number algebra, the algebra of some data structures, and function algebra. In this section we say what is common to all of them.

Order of Evaluation

An algebra consists of expressions, and the expressions consist of operators and operands. Placing operators between operands makes some expressions ambiguous. For example, $2+3\times4$ might mean that 2 and 3 are added, and then the result is multiplied by 4, or that 2 is added to the result of multiplying 3 by 4. To say which is meant, we can use parentheses: either $(2+3)\times4$ or $2+(3\times4)$. To prevent a clutter of parentheses, we decide on an order of evaluation. Here is the order of evaluation of all operators in this paper.

0	constants ⊤ ⊥ 0 1 3.14 and so on	
	variables x y and so on	
	bracketed expressions ( ) { } [ ] ⟨ ⟩ within which the order of evaluation again applies	
1	juxtaposition fx	left to right
2	prefix – ¢ $ ~ 𝒫 𝒟 # ⟶ ∧ ∨ = ǂ § + ×	right to left
	infix ⟶	right to left
	subscript x_n superscript x^n	right to left
3	infix × / ∧ ∨	left to right
4	infix + – ⁺ △ ▽	left to right
5	infix , ,.. ‘ ; ;.. | ◁▷	
6	infix = ǂ < > ≤ ≥ : ∈	

In the order of evaluation, infix + can be found on level 4, and infix × on level 3; that means, in the absence of parentheses, evaluate infix × before infix + . The example 2+3×4 therefore means 2+(3×4) . Within levels 1, 3, and 4 evaluation is from left to right. Within level 2, evaluation is from right to left. x=y=z means (x=y)∧(y=z) and similarly for the other operators and mixtures of operators on level 6.

Format

To help the eye group the symbols properly, it is a good idea to leave space for absent parentheses. The spacing in expression 2 + 3×4 is helpful; the spacing in 2+3 × 4 is misleading.

An expression that is too long to fit on one line must be broken into parts. There are several reasonable ways to do it; here is one suggestion. A long expression in parentheses can be broken at its main operator, which is placed under the opening parenthesis. For example,

(*first part*
+   *second part*   )

A long expression without parentheses can be broken at its main operator, which is placed under where the opening parenthesis belongs. For example,

    first part
=   *second part*

Attention to format makes a big difference in our ability to understand a complex expression.

Expressions and Values

An algebra consists of expressions, which are used to express values in the application domain. For example, the values may be amounts of water, or voltage, or frequency of vibration, or guilt and innocence. We must never use an expression to express more than one value; to do so would be a serious error called inconsistency. Sometimes we may not say what value an expression expresses; that is called incompleteness. For example, we will not be able to determine the value of 0/0 . (I prefer to avoid the question of whether 0/0 has no value, or has a value but we cannot say what it is. It is of no interest whether an expression expresses a value if we cannot determine the value.) Here are four definitions.

Consistency:	at most one value can be determined for each expression
Completeness:	at least one value can be determined for each expression
Expressiveness:	at least one expression can be determined for each value
Uniqueness:	at most one expression can be determined for each value

Consistency is essential; completeness is not. Expressiveness is desirable; uniqueness is not.

As an economy of speech, we say “ $2+3$ has value 5 ” to mean “ $2+3$ has the same value as 5 has”, or “expression $2+3$ expresses the same value as expression 5 expresses”.

Variables and Instantiation

A variable is a kind of expression. In this paper we use single italic letters like x for variables (but that's not a principle of unified algebra). Expressions represent values, and a variable represents an arbitrary value. A variable can be replaced by another expression. Replacing a variable by another expression is called instantiation. Expression $–2$ is called an instance of expression $–x$. Here is how instantiation works.

- We sometimes have to insert parentheses around expressions that are replacing variables in order to maintain the order of evaluation. Expression $–(2+3)$ is an instance of $–x$, but $–2+3$ is not.

- When the same variable occurs more than once in an expression, it must be replaced by the same expression at each occurrence. Expression $2+2$ is an instance of $x+x$, but $2+3$ is not. However, different variables may be replaced by the same or different expressions. Expression $2+2$ is an instance of $x+y$.

Evaluation Rules

Here are the rules to determine the value of expressions.

<u>Direct Rule</u>	An expression may be given a value by physical means, or by other means outside the algebra. This is the way an algebra is applied.
Example:	By marking numbers along a stick we give them length values.
Example:	We might decide to use $\top$ to express truth and $\bot$ to express falsity.
<u>Indirect Rule</u>	An expression may be given a value by saying that it has the same value as another expression whose value is already known. This rule is used in two forms: value tables, and laws.
Example:	In binary algebra, on one of the value tables, from the row labelled $x \wedge y$ and the column labelled $\top\top$, we will see that $\top \wedge \top$ has value $\top$.
Example:	From the first of the common laws, we will see that $x=x$ has value $\top$.
<u>Completion Rule</u>	If the values of some subexpressions of an expression are unknown, and all ways of assigning them values give it the same value, then it has that value.
Example:	In binary algebra, we will see from the value tables that $x \vee –x$ has value $\top$.
Example:	In binary algebra, we will see from the value tables that $x \wedge –x$ has value $\bot$.
<u>Consistency Rule</u>	If it would be inconsistent for an expression to have a particular value, then it has another value. More generally, if it would be inconsistent for several expressions to have a particular assignment of values, then they have another assignment of values.
Example:	In binary algebra, we will see from the value tables that if both x and $x \le y$

	have value $\top$, then so has y .
Example:	In binary algebra, we will see from the value tables that if $-x$ has value $\top$, then x has value $\bot$.
Example:	In binary algebra, we will see from the value tables that if $x=y$ has value $\top$, then x and y have the same value, and if $x=y$ has value $\bot$, then x and y have different values.
<u>Transparency Rule</u>	An expression does not change value when a subexpression is replaced by another with the same value.
Example:	If x and y have the same value, then $x+z$ and $y+z$ have the same value.
<u>Instance Rule</u>	If the value of an expression can be determined, then all its instances have that same value.
Example:	Since $x=x$ has value $\top$, therefore $x + y\times z = x + y\times z$ has value $\top$.

Using the Direct Rule, we apply an algebra, and that gives its symbols an interpretation. Suppose we apply binary algebra by giving both $\top$ and $\bot$ the same value. Although $\top \neq \bot$ is a law, we cannot say that $\top$ and $\bot$ have different values. If we mark numbers on a stick in random places, then we cannot interpret $=$ as "same", $\neq$ as "different", and $>$ as "greater". For the rest of this paper, we assume that $\top$ and $\bot$ have different values, and that numbers are marked in the traditional way, so that $=$, $\neq$, $>$, and all the other symbols have their traditional interpretation.

Binary Algebra

The expressions of binary algebra are called binary expressions. Binary expressions can be used to represent anything that comes in two kinds, such as true and false statements, high and low voltage, satisfactory and unsatisfactory computations, innocent and guilty behavior. In any application of binary algebra, the two things being represented are called the "binary values". For example, in one application the binary values are truth and falsity; in another they are innocence and guilt. Binary expressions include:

$\top$	"top"
$\bot$	"bottom"
$-x$	"negate x "
$x=y$	" x equal y "
$x\neq y$	" x differ y "
$x<y$	" x below y "
$x>y$	" x above y "
$x\leq y$	" x below equal y "
$x\geq y$	" x above equal y "
$x\wedge y$	" x min y "; the top of the symbol is narrow; the symbol doesn't hold water
$x\vee y$	" x max y "; the top of the symbol is wide; the symbol holds water
$x\triangle y$	" x neg min y "
$x\triangledown y$	" x neg max y "
$x\lhd y\rhd z$	" x if y else z "

The two simplest binary expressions are $\top$ and $\bot$. Expression $\top$ represents one binary value, and expression $\bot$ represents the other. In the other binary expressions, the variables x , y , and z may be replaced by any binary expressions. Whichever value is represented by expression x , expression $-x$ represents the other value. This rule can be shown with the aid of a value table.

x	⊤	⊥
$-x$	⊥	⊤

This table says that $-$⊤ represents the same value that ⊥ represents, and that $-$⊥ represents the same value that ⊤ represents. We can similarly show how to evaluate other binary expressions.

xy	⊤⊤	⊤⊥	⊥⊤	⊥⊥
$x=y$	⊤	⊥	⊥	⊤
x⧧*y*	⊥	⊤	⊤	⊥
$x<y$	⊥	⊥	⊤	⊥
$x>y$	⊥	⊤	⊥	⊥
$x\leq y$	⊤	⊥	⊤	⊤
$x\geq y$	⊤	⊤	⊥	⊤
$x\wedge y$	⊤	⊥	⊥	⊥
$x\vee y$	⊤	⊤	⊤	⊥
$x\triangle y$	⊥	⊤	⊤	⊤
$x\triangledown y$	⊥	⊥	⊥	⊤

xyz	⊤⊤⊤	⊤⊤⊥	⊤⊥⊤	⊤⊥⊥	⊥⊤⊤	⊥⊤⊥	⊥⊥⊤	⊥⊥⊥
$x \lhd y \rhd z$	⊤	⊤	⊤	⊥	⊥	⊥	⊤	⊥

Preference

We have two binary values, and so far we have not shown any preference for one over the other. Now we shall show a preference for expressions with the same value as ⊤ in four ways. One way is to abbreviate the statement “Expression x has value ⊤ .” by just writing x , without saying anything about it. Whenever we just write a binary expression, we mean that it has value ⊤ (expresses the same value that ⊤ expresses). For example, instead of saying “Expression ⊤=⊤ has value ⊤ .” we just say “ ⊤=⊤ ”.

Another way we show a preference is by the use of the words “solution” and “law”. A solution to a binary expression is an assignment of values to its variables that gives it the same value as ⊤ ; we have no name for an assignment that gives it the same value as ⊥ . A law is a binary expression for which any assignment of values to its variables gives it the value ⊤ , and so by the Completion Rule it too has value ⊤ ; we have no name for a binary expression that has value ⊥ .

We often use the Indirect Rule by stating that an expression is a law, which means we are assigning it the same value as ⊤ . If we want to assign ⊥ 's value to expression x , instead we state the law $-x$, and then rely on the Consistency Rule to say that x has value ⊥ . In other algebras, if we want to say that x has the same value as y (not a binary value), instead we say that $x=y$ has value ⊤ by stating that $x=y$ is a law.

The final way we show a preference is in the applications of binary algebra. When we apply it to reasoning, we choose to use ⊤ for true statements and ⊥ for false statements. When we use binary expressions as specifications, we choose to use ⊤ for satisfactory objects, and ⊥ for unsatisfactory objects. When we use binary expressions to codify laws, we choose to use ⊤ for innocent behavior and ⊥ for guilty behavior. In each case we could just as well have chosen to use ⊤ and ⊥ the other way round, but the tradition is to use ⊤ for the preferable alternative.

Ternary Algebra

Between the two values represented by $\top$ and $\bot$, we now consider a third value, represented by 0 (pronounced “zero”). Ternary algebra can be applied to anything that comes in three kinds. In one application, the three expressions $\top$, 0, and $\bot$ represent the values “yes”, “maybe”, and “no”. In another, they represent the values “large”, “medium”, and “small”. An assignment of values to variables that gives an expression the value 0 is called a “root” of the expression.

The expressions of ternary algebra, called “ternary expressions”, include all those of binary algebra. To determine the value of these ternary expression, we extend the value tables.

x	$\top$	0	$\bot$
$-x$	$\bot$	0	$\top$

xy	$\top\top$	$\top 0$	$\top\bot$	$0\top$	00	$0\bot$	$\bot\top$	$\bot 0$	$\bot\bot$
$x=y$	$\top$	$\bot$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\bot$	$\top$
$x⧧y$	$\bot$	$\top$	$\top$	$\top$	$\bot$	$\top$	$\top$	$\top$	$\bot$
$x<y$	$\bot$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\top$	$\top$	$\bot$
$x>y$	$\bot$	$\top$	$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\bot$
$x\le y$	$\top$	$\bot$	$\bot$	$\top$	$\top$	$\bot$	$\top$	$\top$	$\top$
$x\ge y$	$\top$	$\top$	$\top$	$\bot$	$\top$	$\top$	$\bot$	$\bot$	$\top$
$x\wedge y$	$\top$	0	$\bot$	0	0	$\bot$	$\bot$	$\bot$	$\bot$
$x\vee y$	$\top$	$\top$	$\top$	$\top$	0	0	$\top$	0	$\bot$
$x\triangle y$	$\bot$	0	$\top$	0	0	$\top$	$\top$	$\top$	$\top$
$x\triangledown y$	$\bot$	$\bot$	$\bot$	$\bot$	0	0	$\bot$	0	$\top$

When the variables have binary values, each expression has the same value as it had in binary algebra; in that sense, we have extended binary algebra to ternary algebra in a consistent way. All our future extensions will likewise be consistent. The expression $x\vee -x$ is a law of binary algebra because both assignments of binary values to variable x give it the value $\top$; but it is not a law of ternary algebra because when x has value 0, $x\vee -x$ has value 0. The expression $x=-x$ has no solution in binary algebra because both assignments of binary values give it the value $\bot$; in ternary algebra its solution is 0. By extending the algebra, we have lost some laws, but gained some solutions.

We can add many new ternary expressions. For example, we can add approximate equality and addition modulo 3 with the value table:

xy	$\top\top$	$\top 0$	$\top\bot$	$0\top$	00	$0\bot$	$\bot\top$	$\bot 0$	$\bot\bot$
$x\approx y$	$\top$	0	$\bot$	0	0	0	$\bot$	0	$\top$
$x\oplus y$	$\bot$	$\top$	0	$\top$	0	$\bot$	0	$\bot$	$\top$

but we do not pursue ternary algebra further. We could now pursue a four-valued algebra, or five-valued algebra, but instead we leap to an infinite-valued algebra. Value tables, which are already becoming cumbersome ($x\lhd y\rhd z$ takes 27 columns in ternary algebra), become impossible with infinitely many values, so from now on we give values to new expressions by stating laws.

Common Laws

We are about to introduce numbers, bunches, sets, strings, lists, and functions. We introduce each of them by saying how to write them and giving their laws. Some of the laws are common to all of them as well as to binary and ternary expressions, so we present them once. They are:

$x = x$	reflexivity
$(x=y) \; = \; (y=x)$	symmetry
$(x=y) \wedge (y=z) \; \le \; (x=z)$	transitivity
$(x \neq y) = -(x=y)$	
$-(x<x)$	irreflexivity
$-((x<y) \wedge (y<x))$	antisymmetry
$(x<y) \wedge (y<z) \; \le \; (x<z)$	transitivity
$(x<y) = (-x>-y)$	reflection
$-((x<y) \wedge (x=y))$	exclusivity
$(x \le y) \; = \; (x<y) \vee (x=y)$	inclusivity
$(x>y) \; = \; (y<x)$	mirror
$(x \ge y) \; = \; (y \le x)$	mirror
$(x \le y) = (x \wedge y = x) = (x \vee y = y)$	
$x \triangle y \; = \; -(x \wedge y)$	
$x \triangledown y \; = \; -(x \vee y)$	
$- -x = x$	self-inverse
$(x \lhd \top \rhd y) = x$	
$(x \lhd \bot \rhd y) = y$	
$x \wedge y \; = \; y \wedge x$	symmetry
$x \vee y \; = \; y \vee x$	symmetry
$x \triangle y \; = \; y \triangle x$	symmetry
$x \triangledown y \; = \; y \triangledown x$	symmetry
$(x \wedge y) \wedge z \; = \; x \wedge (y \wedge z)$	associativity
$(x \vee y) \vee z \; = \; x \vee (y \vee z)$	associativity
$x \wedge x \; = \; x$	idempotence
$x \vee x \; = \; x$	idempotence
$x \wedge z \; \le \; (x \vee y) \wedge (-y \vee z) \; = \; (x \wedge -y) \vee (y \wedge z) \; \le \; x \vee z$	resolution

The following laws show how $-$ distributes over other operators, or can be factored out.

$-(x=y) \; = \; (-x \neq -y)$	$-(x \neq y) \; = \; (-x = -y)$
$-(x<y) \; = \; (-x \le -y)$	$-(x \le y) \; = \; (-x < -y)$
$-(x>y) \; = \; (-x \ge -y)$	$-(x \ge y) \; = \; (-x > -y)$
$-(x \wedge y) \; = \; -x \vee -y$	$-(x \vee y) \; = \; -x \wedge -y$
$-(x \triangle y) \; = \; -x \triangledown -y$	$-(x \triangledown y) \; = \; -x \triangle -y$
$-(x \lhd y \rhd z) \; = \; -x \lhd y \rhd -z$	

It is an interesting mathematical exercise to find a minimal set of laws for an algebra. But those who wish to use the algebra need to know many laws, and to them minimality is of no concern. In this paper, no attention has been paid to minimality.

Number Algebra

We now fill in the spaces between $\top$, 0 , and $\bot$. All operators apply to all values. The ordering places the $\top$ and $\bot$ values at the extremes, with all the other values in between.

$\bot$ ... -3 -2 -1 0 1 2 3 ... $\top$

(In this paper, I consider only real numbers, not complex numbers.)

The expressions of number algebra are called number expressions. Number expressions can be used to represent anything that comes in various quantities, such as apples and water ($\top$ represents an infinite quantity, and $\bot$ represents an infinite deficit). Expressions are formed as follows.

any sequence of one or more decimal digits, such as 5296

any of the ways of forming an expression presented previously, such as

-5296 or $5296\wedge 375$ or $5297=375$

$x+y$	“ x plus y ”
$x-y$	“ x minus y ”
$x\times y$	“ x times y ”
x/y	“ x divided by y ”, “ x over y ”
x^y	“ x to the power y ”

Anyone is welcome to invent new expressions and add them to the list.

Now that we have new expressions, we assign some of them the same value as $\top$. In these laws, d is a sequence of digits.

$d0+1 = d1$	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
$x+0 = x$	identity
$x+y = y+x$	symmetry
$x+(y+z) = (x+y)+z$	associativity
$(\bot<x<\top) \le ((x+y = x+z) = (y=z))$	cancellation
$(\bot<x) \le (\top+x = \top)$	absorption
$(x<\top) \le (\bot+x = \bot)$	absorption
$x + y\wedge z = (x+y) \wedge (x+z)$	distributivity
$x + y\vee z = (x+y) \vee (x+z)$	distributivity
$x + (y\vartriangle z) = (x-y) \vee (x-z)$	
$x + (y\triangledown z) = (x-y) \wedge (x-z)$	
$x + (y \triangleleft z \triangleright w) = x+y \triangleleft z \triangleright x+w$	distributivity
$-x = 0 - x$	
$-(x+y) = -x + -y$	distributivity
$-(x-y) = -x - -y$	distributivity
$-(x\times y) = (-x)\times y$	associativity
$-(x/y) = (-x)/y$	associativity

$x-y = -(y-x)$	antisymmetry
$x-y \; = \; x + -y$	
$x + (y - z) \; = \; (x + y) - z$	associativity
$(\bot<x<\top) \le ((x-y = x-z) = (y=z))$	cancellation
$(\bot<x<\top) \le (x-x = 0)$	inverse
$(x<\top) \le (\top-x = \top)$	absorption
$(\bot<x) \le (\bot-x = \bot)$	absorption
$(\bot<x<\top) \le (x\times 0 = 0)$	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
$(\bot<x<\top) \wedge (x\ne 0) \le ((x\times y = x\times z) = (y=z))$	cancellation
$(0<x) \le (x\times\top = \top)$	absorption
$(0<x) \le (x\times\bot = \bot)$	absorption
$x/1 = x$	identity
$(\bot<x<\top) \wedge (x\ne 0) \le (x/x = 1)$	inverse
$x\times(y/z) = (x\times y)/z$	associativity
$(\bot<x<\top) \le (x/\top = 0 = x/\bot)$	
$(\bot<x<\top) \le (x^0 = 1)$	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
$(\bot<x<\top) \le ((x+y < x+z) = (y<z))$	cancellation, translation
$(0<x<\top) \le ((x\times y < x\times z) = (y<z))$	cancellation, scale
$(x<y) \vee (x=y) \vee (x>y)$	trichotomy
$\bot \le x \le \top$	extremes

Calculation

Given an expression, it is often useful to find a simpler expression with the same value. For example,

	$x\times(z+1) - y\times(z-1) - z\times(x-y)$	distribute
$=$	$(x\times z + x\times 1) - (y\times z - y\times 1) - (z\times x - z\times y)$	unity and double negation
$=$	$x\times z + x - y\times z + y - z\times x + z\times y$	symmetry and associativity
$=$	$x + y + (x\times z - x\times z) + (y\times z - y\times z)$	zero and identity
$=$	$x+y$	

The entire five lines (without the hints that appear to the right) form one binary expression meaning

$$\begin{array}{ll}
 & (x\times(z+1) - y\times(z-1) - z\times(x-y) \; = \; (x\times z + x\times 1) - (y\times z - y\times 1) - (z\times x - z\times y)) \\
\wedge & ((x\times z + x\times 1) - (y\times z - y\times 1) - (z\times x - z\times y) \; = \; x\times z + x - y\times z + y - z\times x + z\times y) \\
\wedge & (x\times z + x - y\times z + y - z\times x + z\times y \; = \; x + y + (x\times z - x\times z) + (y\times z - y\times z)) \\
\wedge & (x + y + (x\times z - x\times z) + (y\times z - y\times z) \; = \; x+y)
\end{array}$$

By simply writing it, we are saying that it has value $\top$. The hint “distribute” is intended to make it clear that

$$x\times(z+1) - y\times(z-1) - z\times(x-y) \; = \; (x\times z + x\times 1) - (y\times z - y\times 1) - (z\times x - z\times y)$$

has value $\top$; the hint “unity and double negation” is intended to make it clear that

$$(x\times z + x\times 1) - (y\times z - y\times 1) - (z\times x - z\times y) \; = \; x\times z + x - y\times z + y - z\times x + z\times y$$

has value $\top$; and so on. By the transitivity of $=$ and the Consistency Rule we see that

$x \times (z+1) - y \times (z-1) - z \times (x-y) \ = \ x+y$
has value $\top$, and so $x \times (z+1) - y \times (z-1) - z \times (x-y)$ and $x+y$ have the same value.

We can use operators other than $=$ down the left side of a calculation, even a mixture, as long as there is transitivity. For example, if x is a real-valued variable,

$$\begin{array}{lll} & x \times (x+2) & \text{distribute} \\ = & x^2 + 2 \times x & \text{identity and zero} \\ = & x^2 + 2 \times x + 1 - 1 & \text{factor} \\ = & (x+1)^2 - 1 & \text{a square is nonnegative} \\ \ge & -1 & \end{array}$$

tells us that $x \times (x+2) \ge -1$ has value $\top$.

The level of hint depends on the knowledge of the intended audience. A hint may refer to some laws, or to a calculation done elsewhere, or to some missing steps that a knowledgeable reader could reasonably be expected to supply.

Variation

One effective way of calculating is to increase or decrease an expression by increasing or decreasing a subexpression. We can increase $x \wedge y$ by increasing y. We can increase $-x$ by decreasing x. As an example, let a and b be binary.

$$\begin{array}{lll} & a \vartriangle (a \triangledown b) & \\ = & -(a \wedge -(a \vee b)) & \text{decrease } a \vee b \text{ to } a \text{ and so decrease the whole expression} \\ \ge & -(a \wedge -a) & \text{use a previous example} \\ = & -\bot & \\ = & \top & \end{array}$$

And so $a \vartriangle (a \triangledown b)$ has a value that is above or equal to $\top$, and since there is nothing above $\top$, its value is $\top$.

Here is a catalogue of variations for use in calculations.

$-x$ varies inversely with x.
$x+y$ varies directly with x and directly with y.
$x-y$ varies directly with x and inversely with y.
$x \wedge y$ varies directly with x and directly with y.
$x \vee y$ varies directly with x and directly with y.
$x \vartriangle y$ varies inversely with x and inversely with y.
$x \triangledown y$ varies inversely with x and inversely with y.
$x<y$ varies inversely with x and directly with y.
$x>y$ varies directly with x and inversely with y.
$x \le y$ varies inversely with x and directly with y.
$x \ge y$ varies directly with x and inversely with y.
$x \lhd y \rhd z$ varies directly with x and directly with z.

Context

Consider an expression of the form $x \wedge y$ where x and y are binary. When we are simplifying x, we can suppose that y has value $\top$. If y really does have value $\top$, then we have done nothing wrong. If y has value $\bot$, then $x \wedge y$ has value $\bot$ no matter which value x has; so no matter how we change x, we don't change the value of $x \wedge y$. For exactly the same reason, we can suppose that x has value $\top$ when we are simplifying y . However, we cannot make both suppositions simultaneously and simplify both x and y at the same time. (If we could, then $x \wedge x$ could be simplified to $\top$.)

Here is an example.

	$(x + x \times y + y = 5) \wedge (x - x \times y + y = 1)$	subtract and add $2 \times x \times y$
$=$	$(x - x \times y + y + 2 \times x \times y = 5) \wedge (x - x \times y + y = 1)$	use second part to simplify first
$=$	$(1 + 2 \times x \times y = 5) \wedge (x - x \times y + y = 1)$	simplify
$=$	$(2 \times x \times y = 4) \wedge (x - x \times y + y = 1)$	simplify
$=$	$(x \times y = 2) \wedge (x - x \times y + y = 1)$	use first part to simplify second
$=$	$(x \times y = 2) \wedge (x - 2 + y = 1)$	simplify
$=$	$(x \times y = 2) \wedge (x+y = 3)$	
$\ge$	$(x=1) \wedge (y=2)$	

We can generalize this sort of reasoning to apply to number expressions.

In $x<y$,	when simplifying x , we can suppose $y ⧧ \bot$;
	when simplifying y , we can suppose $x ⧧ \top$.
In $x>y$,	when simplifying x , we can suppose $y ⧧ \top$;
	when simplifying y , we can suppose $x ⧧ \bot$.
In $x \le y$,	when simplifying x , we can suppose $y ⧧ \top$;
	when simplifying y , we can suppose $x ⧧ \bot$.
In $x \ge y$,	when simplifying x , we can suppose $y ⧧ \bot$;
	when simplifying y , we can suppose $x ⧧ \top$.
In $x \wedge y$,	when simplifying x , we can suppose $y ⧧ \bot$;
	when simplifying y , we can suppose $x ⧧ \bot$.
In $x \vee y$,	when simplifying x , we can suppose $y ⧧ \top$;
	when simplifying y , we can suppose $x ⧧ \top$.
In $x \vartriangle y$,	when simplifying x , we can suppose $y ⧧ \bot$;
	when simplifying y , we can suppose $x ⧧ \bot$.
In $x \triangledown y$,	when simplifying x , we can suppose $y ⧧ \top$;
	when simplifying y , we can suppose $x ⧧ \top$.
In $x \lhd y \rhd z$,	when simplifying x , we can suppose $y ⧧ \bot$;
	when simplifying z , we can suppose $y ⧧ \top$.

Data Structures

A data structure is a collection, or aggregate, of data. The kinds of structuring we consider are packaging and indexing. These two kinds of structure give us four data structures.

unpackaged, unindexed:	bunch
packaged, unindexed:	set
unpackaged, indexed:	string
packaged, indexed:	list

Bunches

A bunch represents a collection of objects. For contrast, a set represents a collection of objects in a package or container. The contents of a set is a bunch. These vague descriptions are made precise as follows.

Any binary or number (and later also set, string of elements, and list of elements) is an elementary bunch, or element. For example, the number 2 is an elementary bunch, or synonymously, an element. Indeed, every expression is a bunch expression, though not all are elementary.

If A and B are bunches, then

A , B	“ A union B ”
$A ‘ B$	“ A intersection B ”

are bunches,

$¢A$	“size of A ”

is a number, and

$A\colon B$	“ A is in B ”, “ A is included in B ”

is a binary expression.

The size of a bunch is the number of elements it includes. Elements are bunches of size 1 .

$¢2 = 1$
$¢(0, 2, 5, 9) = 4$

Here are three quick examples of bunch inclusion.

$2\colon 0, 2, 5, 9$
$2\colon 2$
$2, 9\colon 0, 2, 5, 9$

The first says that 2 is in the bunch consisting of 0, 2, 5, 9 . The second says that 2 is in the bunch consisting of only 2 . Note that we do not say “a bunch contains its elements”, but rather “a bunch consists of its elements”. The third example says that both 2 and 9 are in 0, 2, 5, 9 , or in other words, the bunch 2, 9 is included in the bunch 0, 2, 5, 9 .

Here are the bunch laws. In these laws, x and y are elements (elementary bunches), and A , B , and C are arbitrary bunches.

$(x\colon y) \;=\; (x=y)$	elementary law
$(x\colon A,B) \;=\; (x\colon A) \;\vee\; (x\colon B)$	compound law
$A,A = A$	idempotence
$A,B = B,A$	symmetry
$A,(B,C) = (A,B),C$	associativity
$A‘A = A$	idempotence
$A‘B = B‘A$	symmetry
$A‘(B‘C) = (A‘B)‘C$	associativity
$(A,B\colon C) \;=\; (A\colon C) \;\wedge\; (B\colon C)$	
$(A\colon B‘C) \;=\; (A\colon B) \;\wedge\; (A\colon C)$	
$A\colon A,B$	generalization
$A‘B\colon A$	specialization
$A\colon A$	reflexivity
$(A\colon B) \;\wedge\; (B\colon A) \;=\; (A=B)$	antisymmetry
$(A\colon B) \;\wedge\; (B\colon C) \;\le\; (A\colon C)$	transitivity
$¢x = 1$	
$¢(A, B) + ¢(A‘B) = ¢A + ¢B$	
$-(x\colon A) \;\le\; (¢(A‘x) = 0)$	
$(A\colon B) \;\le\; (¢A \le ¢B)$	

For other laws see [2].

Here are several bunches that are useful enough to be named:

null			the empty bunch
bin	=	$\top, \bot$	the binary values
nat	=	0, 1, 2, ...	the natural numbers
int	=	..., –2, –1, 0, 1, 2, ...	the integer numbers
rat	=	0, –1, 2/3, ...	the rational numbers
real			the real numbers

We define them formally in a moment.

The operators , ‘ ¢ : = ⧣ ⊲⊳ apply to bunch operands according to the axioms already presented. Other operators can be applied to bunches with the understanding that they apply to the elements of the bunch. In other words, they distribute over bunch union. For example,

$$-null \;=\; null$$
$$-(A, B) \;=\; -A, -B$$
$$A+null \;=\; null$$
$$A+(B, C) \;=\; A+B, A+C$$

This makes it easy to express the positive naturals $(nat+1)$, the even naturals $(nat\times 2)$, the squares (nat^2), the powers of two (2^{nat}), and many other things.

We define the empty bunch, *null*, by the laws

$$null\colon A$$
$$(¢A = 0) \;=\; (A = null)$$

The bunch *bin* is defined by the law $bin = \top, \bot$.

The bunch *nat* is defined by two laws.

$0, nat+1\colon nat$	construction
$(0, B+1\colon B) \;\le\; (nat\colon B)$	induction

The first, construction, says that 0, 1, 2, and so on, are in *nat* . The second, induction, says that nothing else is in *nat* by saying that of all the bunches satisfying the construction law, *nat* is the smallest. Now that we have *nat* , we can define *int* and *rat* as follows:

$$int \;=\; nat, -nat$$
$$rat \;=\; int/(nat+1)$$

The law defining *real* will be given later in the section titled “Limits”.

We also use the notation

$m,..n$	“ m to n ”

where m and n are integer or binary and $m\le n$. For integer m and n , this notation means the bunch $m, m+1, m+2, ..., n-1$. The asymmetric notation is a reminder that the left end is included but the right end is excluded. Its law is

$$(x\colon m,..n) \;=\; (x\colon int) \wedge (m\le x<n)$$

For example,

$$0,..3 \;=\; 0, 1, 2$$
$$0,..0 \;=\; null$$
$$¢(m,..n) \;=\; n-m$$

Sets

Let A be any bunch (anything). Then

$\{A\}$	“set containing A”

is a set. Thus $\{null\}$ is the empty set, and the set containing the first three natural numbers is expressed as $\{0, 1, 2\}$ or as $\{0,..3\}$, and $\{nat\}$ is the set of natural numbers. All sets are elements; not all bunches are elements; that is the difference between sets and bunches. We can form the bunch $1, \{3, 7\}$ consisting of two elements, and from it the set $\{1, \{3, 7\}\}$ containing two elements, and in that way we build a structure of nested sets. Set formation has an inverse. If S is any set, then

$\sim S$	“contents of S”

is its contents. For example,

$$\sim\{0, 1\} \;=\; 0, 1$$

Now that we have bunches, the laws of sets are very easily stated.

$\{A\} \;⧧\; A$	structure
$\sim\{A\} \;=\; A$	contents
$\#\{A\} = ¢A$	size
$(A \in \{B\}) \;=\; (A{:}\, B)$	elements
$(\{A\} \le \{B\}) \;=\; (A{:}\, B)$	subset
$(\{A\} \in \mathcal{P}\{B\}) \;=\; (A{:}\, B)$	powerset
$\{A\} \vee \{B\} \;=\; \{A, B\}$	union
$\{A\} \wedge \{B\} \;=\; \{A \text{ ‘ } B\}$	intersection
$(\{A\} = \{B\}) \;=\; (A = B)$	equation

Note that the element, subset, and powerset laws are all just bunch inclusion.

Strings

Just as bunches and sets are, respectively, unpackaged and packaged collections, so strings and lists are, respectively, unpackaged and packaged sequences. There are sets of sets, and lists of lists, but there are neither bunches of bunches nor strings of strings.

The simplest string is

nil	the empty string

Any binary, number, set (and later also list and function) is a one-item string, or item. For example, the number 2 is a one-item string, or item. A bunch of items is also an item. Strings are catenated (joined) together by semicolons to make longer strings. For example,

$$4; 2; 4; 6$$

is a four-item string. The length of a string is the number of items, and is obtained by the $\$$ operator.

$$\$(4; 2; 4; 6) \;=\; 4$$

The index of an item is the number of items that precede it. In other words, indexing is from 0 . An index is not an arbitrary label, but a measure of how much has gone before. Your life begins at year 0 , a highway begins at mile 0 , and so on. We refer to the items in a string as “item 0”, “item 1”, “item 2”, and so on; we never say “the third item” due to the possible confusion between item 2 and item 3. We obtain an item of a string by subscripting. For example,

$$(3; 5; 7; 9)_2 \;=\; 7$$

In general, S_n is item n of string S . We can even pick out a whole string of items, as in the following example.

$(3; 5; 7; 9)_{2; 1; 2} = 7; 5; 7$

Strings can be compared for equality and order. To be equal, strings must be of equal length, and have equal items at each index. The order of two strings is determined by the items at the first index where they differ. For example,

$3; 6; 4; 7 < 3; 7; 2$

If there is no index where they differ, the shorter string comes before the longer one.

$3; 6; 4 < 3; 6; 4; 7$

This ordering is known as lexicographic order; it is the ordering used in dictionaries.

Here is the syntax of strings. If i is an item, and S and T are strings, then

nil	the empty string
i	an item
$S;T$	“ S catenate T ”
S_T	“ S sub T ”
$S \wedge T$	“ S min T ”
$S \vee T$	“ S max T ”

are strings, and

$\$S$	“length of S ”

is a natural number or $\top$, and

$S=T$	“ S equals T ”
$S \neq T$	“ S differs from T ”
$S<T$	“ S is less than T ”
$S>T$	“ S is greater than T ”
$S \le T$	“ S is at most T ”
$S \ge T$	“ S is at least T ”

are binary.

Here are the laws of string algebra. In these laws, S , T , and U are strings, and i and j are items.

$nil; S = S; nil = S$

$S; (T; U) = (S; T); U$

$\$nil = 0$

$\$i = 1$

$\$(S; T) = \$S + \$T$

$(S; i; T)_{\$S} = i$

$S_{(T_U)} = (S_T)_U$

$(i=j) = (S; i; T = S; j; T)$

$(i<j) \le (S; i; T < S; j; U)$

$nil \le S < S; i; T$

$S_{null} = null$

$S_{T, U} = S_T , S_U$

$S_{\{T\}} = \{S_T\}$

$S_{nil} = nil$

$S_{T; U} = S_T ; S_U$

We also use the notation

$x;..y$	“ x to y ” (same pronunciation as $x,..y$)

where x and y are integers or binary and $x \le y$. As in the similar bunch notation, x is included and y excluded, so that

$\$(x;..y) = y-x$

Here are the laws.

$x;..x = nil$

$x;..x+1 \ = \ x$
$(x;..y) \ ; \ (y;..z) \ = \ x;..z$

String catenation distributes over bunch union:
$A;\ null;\ B \ = \ null$
$(A,\ B);\ (C,\ D) \ = \ A;C,\ A;D,\ B;C,\ B;D$
So a string of bunches is equal to a bunch of strings. Thus, for example,
$0;\ 1;\ 2: \quad nat;\ 1;\ (0,..10)$
because $0: nat$ and $1: 1$ and $2: 0,..10$. A string is an element (elementary bunch) just when all its items are elements; so $0;1;2$ is an element, but $nat;\ 1;\ (0,..10)$ is not.

Our main purpose in presenting string algebra is as a stepping stone to the presentation of list algebra.

Lists

A list is a packaged string. For example,
$[0;\ 1;\ 2]$
is a list of three items. List brackets [ ] distribute over bunch union.
$[null] \ = \ null$
$[A,\ B] \ = \ [A],\ [B]$
Because of the distribution we can say
$[0;\ 1;\ 2]: \quad [nat;\ 1;\ (0,..10)]$
On the left of the colon we have a list of integers; on the right we have a list of bunches, or equivalently, a bunch of lists. A list is an element (elementary bunch) just when all its items are elements; $[0;\ 1;\ 2]$ is an element, but $[nat;\ 1;\ (0,..10)]$ is not.

Here is the syntax of lists. Let S be a string, L and M be lists, n be a natural number, and i be an item. Then

$[S]$	“list containing S”
$L\ M$	“$L\ M$” or “L composed with M”
L^+M	“L catenate M”
$n\to i \mid L$	“n maps to i otherwise L”
$L\wedge M$	“L min M”
$L\vee M$	“L max M”

are lists,

$L\ n$	“$L\ n$” or “L index n”

is an item,

$\#L$	“length of L”

is a natural number or binary, and

$L=M$	“L equals M”
$L\neq M$	“L differs from M”
$L<M$	“L is less than M”
$L>M$	“L is greater than M”
$L\le M$	“L is at most M”
$L\ge M$	“L is at least M”

are binary.

Parentheses may be used around any expression, so we may write $L(n)$. If the index is not simple, we will have to enclose it in parentheses. When there is no danger of confusion, we may write Ln without a space between, but when we use multicharacter names, we must put a space between.

The length of a list is the number of items it contains.

$\#[3;\ 5;\ 7;\ 4] \ = \ 4$

List indexes, like string indexes, start at 0 . An item can be selected from a list by juxtaposing (placing next to each other) a list and an index.

$[3;\ 5;\ 7;\ 4]\ 2 \ = \ 7$

A list of indexes gives a list of selected items. For example,

$[3;\ 5;\ 7;\ 4]\ [2;\ 1;\ 2] \ = \ [7;\ 5;\ 7]$

This is called list composition. List catenation is written with a small raised plus sign $^+$.

$[3;\ 5;\ 7;\ 4]^+[2;\ 1;\ 2] \ = \ [3;\ 5;\ 7;\ 4;\ 2;\ 1;\ 2]$

The notation $n \to i \mid L$ gives us a list just like L except that item n is i .

$2 \to 22 \mid [10;..15] \ = \ [10;\ 11;\ 22;\ 13;\ 14]$

$2 \to 22 \mid 3 \to 33 \mid [10;..15] \ = \ [10;\ 11;\ 22;\ 33;\ 14]$

Let $L \ = \ [10;..15]$. Then

$2 \to L3 \mid 3 \to L2 \mid L \ = \ [10;\ 11;\ 13;\ 12;\ 14]$

The order operators $< \le > \ge$ apply to lists; the order is lexicographic, just like string order.

Here are the laws. Let S and T be strings, and let i and j be items.

$\#[S] \ = \ \$S$	length
$[S]^+[T] \ = \ [S;\ T]$	catenation
$[S]\ T \ = \ S_T$	indexing
$[S_T] \ = \ S_{[T]}$	
$[S]\ [T] \ = \ [S_T]$	composition
$(\$S) \to i \mid [S;\ j;\ T] \ = \ [S;\ i;\ T]$	modification
$([S] = [T]) \ = \ (S = T)$	equation
$([S] < [T]) \ = \ (S < T)$	order

Let L , M , and N be lists, and n be natural. Then

$(L\ M)\ n \ = \ L\ (M\ n)$	
$(L\ M)\ N \ = \ L\ (M\ N)$	associativity
$L\ (M^+N) \ = \ L\ M\ ^+\ L\ N$	distributivity

When a list is indexed by a structure, the result will have the same structure. Here is a fancy example. Let $L = [10;\ 11;\ 12]$. Then

$L\ [0,\ \{1,\ [2;\ 1];\ 0\}] \ = \ [L\ 0,\ \{L\ 1,\ [L\ 2;\ L\ 1];\ L\ 0\}] \ = \ [10,\ \{11,\ [12;\ 11];\ 10\}]$

Lists can be items in a list. For example, let

$$A \ = \ [\ [6;\ 3;\ 7;\ 0]\ ; \quad [4;\ 9;\ 2;\ 5]\ ; \quad [1;\ 5;\ 8;\ 3]\ ]$$

Then A is a 2-dimensional array, or more particularly, a 3×4 array. Indexing A with one index gives a list

$A\ 1 \ = \ [4;\ 9;\ 2;\ 5]$

which can then be indexed again to give a number.

$A\ 1\ 2 \ = \ 2$

Functions

A function introduces a local variable with two expressions called the domain and result. It is written in the following form:

$\langle variable\text{: } domain \rightarrow result\rangle$

The scope of the variable begins at the opening angle bracket and extends to the closing angle bracket. All the laws in the context of the function that do not mention the variable are applicable within the function, and the additional law *variable*: *domain* is also applicable within the function. For example, the successor function

$\langle n\text{: } nat \rightarrow n+1\rangle$

introduces local variable n with domain nat and result $n+1$.

As a short form, we can omit the domain and its preceding colon when the domain is known or irrelevant. For example, suppose the surrounding commentary has made it clear that the domain is nat . Then we can write the successor function in the preceding paragraph as

$\langle n \rightarrow n+1\rangle$

When the result of a function does not depend on its variable, we can omit the variable along with the angle brackets and colon as another short form. For example, the constant function

$\langle n\text{: } nat \rightarrow 1\rangle$

can be written more briefly as

$nat{\rightarrow}1$

Finally, if the result does not depend on the variable and the domain is known or irrelevant, we can omit both the variable (and angle brackets) and domain (and preceding colon). For example, the preceding constant function can be written

${\rightarrow}1$

The result of a function can be a function, for example

$\langle d\text{: } nat+1 \rightarrow \langle n\text{: } nat \rightarrow n\text{: } d{\times}nat\rangle\rangle$

This can be called a function of two variables, saying whether its first operand divides its second. Here is a function of two variables in which the first variable is used in the domain of the second.

$\langle n\text{: } nat \rightarrow \langle m\text{: } (0,..n) \rightarrow m{\times}n + n\rangle\rangle$

The constant function of two natural variables

$\langle n\text{: } nat \rightarrow \langle m\text{: } nat \rightarrow 0\rangle\rangle$

can be abbreviated

$nat \rightarrow nat \rightarrow 0$

or, if we know the domains from the surrounding commentary,

${\rightarrow}{\rightarrow}0$

A function introduces a variable that is local to the function. Those variables that appear in the function, and are not introduced by the function, are nonlocal to the function. Any expression may be a part of a larger expression, and so a variable that is nonlocal to a function may be the local variable of a larger enclosing function, or of a smaller enclosed function. Similarly, a variable that is local to a function may be a nonlocal variable of a larger enclosing function, or of a smaller enclosed function.

The formal way to introduce a variable into an expression is the function, and the formal way to eliminate a variable is function application; in other words, function application expresses instantiation. Function f is applied to (operates on) an element x of its domain by the notation $f\,x$, which is pronounced “ f applied to x ” or “ f of x ”. For example

$\langle x\text{: } nat \to x{+}y\rangle\ 3$

is a function application, and it expresses (has the same value as) the instantiation that replaces x in $x{+}y$ with 3 . Here is the Application Law. If x is an element of D , then

$\langle x\text{: } D \to R\rangle\ x\ =\ R$

So, replacing x with 3 , D with nat , and R with $x{+}y$,

$\langle x\text{: } nat \to x{+}y\rangle\ 3\ =\ 3{+}y$

Here is another example.

$$\begin{array}{lll} & \langle d\text{: } nat{+}1 \to \langle n\text{: } nat \to n\text{: } d{\times}nat\rangle\rangle\ 3\ 5 & \text{apply, since } 3\text{: } nat{+}1 \\ = & \langle n\text{: } nat \to n\text{: } 3{\times}nat\rangle\ 5 & \text{apply, since } 5\text{: } nat \\ = & 5\text{: } 3{\times}nat & \\ = & \bot & \end{array}$$

Here is a function that can be applied to a variable number of operands.

$eat\ =\ \langle n\text{: } nat \to 0 \lhd n{=}0 \rhd eat\rangle$

The function eat eats operands until it is fed 0 , whereupon its result is 0 .

Instantiation was introduced near the beginning of this paper, and there were two points explaining how it works; now there are two more.

- Except when instantiating the Application Law, instantiation replaces nonlocal variables only. If we instantiate $\langle x\text{: } nat \to x{+}y\rangle\ (x{+}y)$ by replacing x with y we obtain $\langle x\text{: } nat \to x{+}y\rangle\ (y{+}y)$.

- Except when instantiating the Application Law, instantiation must not place a nonlocal variable where it will appear to be local. We cannot instantiate $\langle x\text{: } nat \to x{+}y\rangle\ (x{+}y)$ by replacing y with x .

The exceptions are due to the fact that the Application Law expresses instantiation.

Aside Why do I have all the words “instantiation”, “application”, “substitution”, “replacement”? Surely I should choose one and stick to it. *End of Aside*

The domain of a function (domain of its variable) is obtained by the $\mathcal{D}$ operator with the Domain Law:

$\mathcal{D}\langle x\text{: } D \to Rx\rangle = D$

When we instantiate the Domain Law, the instantiation rules prevent us from replacing D with an expression in which variable x is nonlocal.

The Extension Law says:

$\langle x\text{: } \mathcal{D}f \to fx\rangle = f$

This law can be instantiated by replacing f with $\langle y\text{: } D \to fy\rangle$ to obtain

$\langle x\text{: } D \to \langle y\text{: } D \to fy\rangle\ x\rangle = \langle y\text{: } D \to fy\rangle$

and if x is an element in D then we can apply the first $\langle y\text{: } D \to fy\rangle$ to x to obtain

$\langle x\text{: } D \to fx\rangle = \langle y\text{: } D \to fy\rangle$

which says that a function in variable x equals a function in variable y obtained by replacing x with y in the result expression. This is called “renaming the variable”. Sometimes renaming is required to allow an instantiation without making a nonlocal variable appear local.

The size of a function is the size of its domain.

$\#f\ =\ ¢\mathcal{D}f$

A function can be conditional, and so can its operand.

$$(f \lhd b \rhd g)\, x \;=\; fx \lhd b \rhd gx$$
$$f\,(x \lhd b \rhd y) \;=\; fx \lhd b \rhd fy$$

A function can be a bunch union,

$$(f, g)\, x \;=\; fx, gx$$

and so can its operand.

Function application can be extended to non-element operands in either of two ways. For each function f, we can choose whether to apply the function even though the operand is non-elementary, or to distribute the application over the operand as follows:

$$f\, null \;=\; null$$
$$f\,(a, b) \;=\; f\,a, f\,b$$

But we cannot choose both options for the same function.

Operators on Functions

The operators $\wedge\ \vee\ +\ \times$ have been defined for two binary or number operands. Following Curry, we now define them for one function operand. $\wedge f$ is the minimum of f. $\vee f$ is the maximum of f. $+f$ is the sum of f. $\times f$ is the product of f. At the same time we define a new operator $\S$ on one function operand: $\S f$ ("solutions of f", or "those f") is the values in the domain of f such that the corresponding result is $\top$. Here are the laws, in which e is an element.

$$\wedge\langle x\colon null \to fx\rangle \;=\; \top \;=\; \wedge(A \to \top)$$
$$\wedge\langle x\colon e \to fx\rangle \;=\; fe$$
$$\wedge\langle x\colon A,B \to fx\rangle \;=\; \wedge\langle x\colon A \to fx\rangle \wedge \wedge\langle x\colon B \to fx\rangle$$
$$\wedge\langle x\colon \S f \to gx\rangle \;=\; \wedge\langle x\colon \mathcal{D}f \to gx \lhd fx \rhd \top\rangle$$

$$\vee\langle x\colon null \to fx\rangle \;=\; \bot \;=\; \vee(A \to \bot)$$
$$\vee\langle x\colon e \to fx\rangle \;=\; fe$$
$$\vee\langle x\colon A,B \to fx\rangle \;=\; \vee\langle x\colon A \to fx\rangle \vee \vee\langle x\colon B \to fx\rangle$$
$$\vee\langle x\colon \S f \to gx\rangle \;=\; \vee\langle x\colon \mathcal{D}f \to gx \lhd fx \rhd \bot\rangle$$

$$+\langle x\colon null \to fx\rangle \;=\; 0 \;=\; +(A \to 0)$$
$$+\langle x\colon e \to fx\rangle \;=\; fe$$
$$+\langle x\colon A,B \to fx\rangle + +\langle x\colon A`B \to fx\rangle \;=\; +\langle x\colon A \to fx\rangle + +\langle x\colon B \to fx\rangle$$
$$+\langle x\colon \S f \to gx\rangle \;=\; +\langle x\colon \mathcal{D}f \to gx \lhd fx \rhd 0\rangle$$

$$\times\langle x\colon null \to fx\rangle \;=\; 1 \;=\; \times(A \to 1)$$
$$\times\langle x\colon e \to fx\rangle \;=\; fe$$
$$\times\langle x\colon A,B \to fx\rangle \times \times\langle x\colon A`B \to fx\rangle \;=\; \times\langle x\colon A \to fx\rangle \times \times\langle x\colon B \to fx\rangle$$
$$\times\langle x\colon \S f \to gx\rangle \;=\; \times\langle x\colon \mathcal{D}f \to gx \lhd fx \rhd 1\rangle$$

$$\S\langle x\colon null \to fx\rangle \;=\; null \;=\; \S(A \to \bot)$$
$$\S\langle x\colon e \to fx\rangle \;=\; e \lhd fe \rhd null$$
$$\S\langle x\colon A,B \to fx\rangle \;=\; \S\langle x\colon A \to fx\rangle\, ,\, \S\langle x\colon B \to fx\rangle$$
$$\S\langle x\colon \S f \to gx\rangle \;=\; \S\langle x\colon \mathcal{D}f \to gx \lhd fx \rhd \bot\rangle$$

$$\S(A \rightarrow \top) \;=\; A$$
$$\S\langle x\text{: } A\text{‘}B \rightarrow fx\rangle \;=\; \S\langle x\text{: } A \rightarrow fx\rangle \text{ ‘ } \S\langle x\text{: } B \rightarrow fx\rangle$$
$$\S\langle x\text{: } A \rightarrow fx\rangle \text{ , } \S\langle x\text{: } A \rightarrow gx\rangle \;=\; \S\langle x\text{: } A \rightarrow fx\vee gx\rangle$$
$$\S\langle x\text{: } A \rightarrow fx\rangle \text{ ‘ } \S\langle x\text{: } A \rightarrow gx\rangle \;=\; \S\langle x\text{: } A \rightarrow fx\wedge gx\rangle$$
$$\wedge\langle x\text{: } \mathcal{D}f \rightarrow (x\text{:}\S f) = fx\rangle$$
$$(x\text{:}\S f) \;=\; (x\text{: } \mathcal{D}f) \wedge fx$$
$$\wedge\langle x\text{: } \S f \rightarrow fx\rangle$$

$$(\wedge(A \rightarrow e) = e) \;\ge\; (A \text{ ǂ } null)$$
$$(\vee(A \rightarrow e) = e) \;\ge\; (A \text{ ǂ } null)$$
$$\S(A \rightarrow e) \;=\; A \lhd e \rhd null$$

$$\wedge\langle x\text{: } A \rightarrow x\text{: } B\rangle \;=\; (A\text{: } B)$$
$$\vee\langle x\text{: } A \rightarrow x\text{: } B\rangle \;=\; (A\text{‘}B \text{ ǂ } null)$$
$$\S\langle x\text{: } A \rightarrow x\text{: } B\rangle \;=\; (A\text{‘}B)$$

If we have decided to distribute application of function f over bunch union, then we also distribute its application over solutions, as follows:

$$f(\S\langle x\text{: } A \rightarrow gx\rangle) \;=\; \S\langle y\text{: } fA \rightarrow \vee\langle x\text{: } A \rightarrow fx=y \wedge gx\rangle\rangle$$

Let f be a function with a non-null domain. Let g be any function. If gx varies directly with x , then

$$\wedge\langle x\text{: } \mathcal{D}f \rightarrow g(fx)\rangle \;\ge\; g(\wedge\langle x\text{: } \mathcal{D}f \rightarrow fx\rangle) \;=\; g(\wedge f)$$
$$\vee\langle x\text{: } \mathcal{D}f \rightarrow g(fx)\rangle \;\le\; g(\vee\langle x\text{: } \mathcal{D}f \rightarrow fx\rangle) \;=\; g(\vee f)$$

If gx varies inversely with x , then

$$\wedge\langle x\text{: } \mathcal{D}f \rightarrow g(fx)\rangle \;\ge\; g(\vee\langle x\text{: } \mathcal{D}f \rightarrow fx\rangle) \;=\; g(\vee f)$$
$$\vee\langle x\text{: } \mathcal{D}f \rightarrow g(fx)\rangle \;\le\; g(\wedge\langle x\text{: } \mathcal{D}f \rightarrow fx\rangle) \;=\; g(\wedge f)$$

And in most instances, $\ge$ and $\le$ can be replaced by $=$. Here are the distributive or factoring laws (omitting the non-null domain).

$$\wedge\langle x \rightarrow -fx\rangle \;=\; -\vee\langle x \rightarrow fx\rangle \;=\; -\vee f$$
$$\wedge\langle x \rightarrow fx + y\rangle \;=\; \wedge\langle x \rightarrow fx\rangle + y \;=\; \wedge f + y$$
$$\wedge\langle x \rightarrow fx - y\rangle \;=\; \wedge\langle x \rightarrow fx\rangle - y \;=\; \wedge f - y$$
$$\wedge\langle x \rightarrow y - fx\rangle \;=\; y - \vee\langle x \rightarrow fx\rangle \;=\; y - \vee f$$
$$\wedge\langle x \rightarrow fx \wedge y\rangle \;=\; \wedge\langle x \rightarrow fx\rangle \wedge y \;=\; \wedge f \wedge y$$
$$\wedge\langle x \rightarrow fx \vee y\rangle \;=\; \wedge\langle x \rightarrow fx\rangle \vee y \;=\; \wedge f \vee y$$
$$\wedge\langle x \rightarrow fx \vartriangle y\rangle \;=\; \vee\langle x \rightarrow fx\rangle \vartriangle y \;=\; \vee f \vartriangle y$$
$$\wedge\langle x \rightarrow fx \triangledown y\rangle \;=\; \vee\langle x \rightarrow fx\rangle \triangledown y \;=\; \vee f \triangledown y$$
$$\wedge\langle x \rightarrow fx < y\rangle \;\ge\; (\vee\langle x \rightarrow fx\rangle < y) \;=\; (\vee f < y)$$
$$\wedge\langle x \rightarrow fx > y\rangle \;\ge\; (\wedge\langle x \rightarrow fx\rangle > y) \;=\; (\wedge f > y)$$
$$\wedge\langle x \rightarrow fx \le y\rangle \;=\; (\vee\langle x \rightarrow fx\rangle \le y) \;=\; (\vee f \le y)$$
$$\wedge\langle x \rightarrow fx \ge y\rangle \;=\; (\wedge\langle x \rightarrow fx\rangle \ge y) \;=\; (\wedge f \ge y)$$
$$\wedge\langle x \rightarrow fx \lhd y \rhd z\rangle \;=\; \wedge\langle x \rightarrow fx\rangle \lhd y \rhd z \;=\; \wedge f \lhd y \rhd z$$

$$\vee\langle x \rightarrow -fx\rangle \;=\; -\wedge\langle x \rightarrow fx\rangle \;=\; -\wedge f$$
$$\vee\langle x \rightarrow fx + y\rangle \;=\; \vee\langle x \rightarrow fx\rangle + y \;=\; \vee f + y$$
$$\vee\langle x \rightarrow fx - y\rangle \;=\; \vee\langle x \rightarrow fx\rangle - y \;=\; \vee f - y$$
$$\vee\langle x \rightarrow y - fx\rangle \;=\; y - \wedge\langle x \rightarrow fx\rangle \;=\; y - \wedge f$$
$$\vee\langle x \rightarrow fx \wedge y\rangle \;=\; \vee\langle x \rightarrow fx\rangle \wedge y \;=\; \vee f \wedge y$$
$$\vee\langle x \rightarrow fx \vee y\rangle \;=\; \vee\langle x \rightarrow fx\rangle \vee y \;=\; \vee f \vee y$$

$$
\begin{array}{lllll}
\vee\langle x \to fx \vartriangle y\rangle & = & \wedge\langle x \to fx\rangle \vartriangle y & = & \wedge f \vartriangle y \\
\vee\langle x \to fx \triangledown y\rangle & = & \wedge\langle x \to fx\rangle \triangledown y & = & \wedge f \triangledown y \\
\vee\langle x \to fx < y\rangle & = & (\wedge\langle x \to fx\rangle < y) & = & (\wedge f < y) \\
\vee\langle x \to fx > y\rangle & = & (\vee\langle x \to fx\rangle > y) & = & (\vee f > y) \\
\vee\langle x \to fx \le y\rangle & \le & (\wedge\langle x \to fx\rangle \le y) & = & (\wedge f \le y) \\
\vee\langle x \to fx \ge y\rangle & \le & (\vee\langle x \to fx\rangle \ge y) & = & (\vee f \ge y) \\
\vee\langle x \to fx \lhd y \rhd z\rangle & = & \vee\langle x \to fx\rangle \lhd y \rhd z & = & \vee f \lhd y \rhd z
\end{array}
$$

Function Inclusion

Consider a function in which the result is a bunch: each element of the domain is mapped to zero or more elements of the range. For example,

$$\langle n\colon nat \to n, n+1\rangle$$

maps each natural number to two natural numbers. Application works as usual:

$$\langle n\colon nat \to n, n+1\rangle\ 3 \quad = \quad 3, 4$$

Functions are sometimes classified as partial or total, and sometimes as deterministic or nondeterministic. If these classifications are thought to be useful, here is one way (although nonstandard) to define them.

partial	sometimes produces no result
total	always produces at least one result
deterministic	always produces at most one result
nondeterministic	sometimes produces more than one result

By these definitions, here is a function that is both partial and nondeterministic.

$$\langle n\colon nat \to (0,..n)\rangle$$

A function f is included in a function g according to the Function Inclusion Law:

$$(f\colon g) \quad = \quad (\mathcal{D}g\colon \mathcal{D}f)\ \wedge\ \wedge\langle x\colon \mathcal{D}g \to fx\colon gx\rangle$$

Using it both ways round, we find function equality is as follows:

$$(f = g) \quad = \quad (\mathcal{D}f = \mathcal{D}g)\ \wedge\ \wedge\langle x\colon \mathcal{D}f \to fx = gx\rangle$$

Let suc be the successor function on the naturals.

$$suc \quad = \quad \langle n\colon nat \to n+1\rangle$$

We now evaluate $suc\colon nat{\to}nat$. Function $nat{\to}nat$ is an abbreviation of $\langle n\colon nat \to nat\rangle$, which has an unused variable. It is a nondeterministic function whose result, for each element of its domain nat , is the bunch nat .

$$
\begin{array}{lll}
 & (suc\colon nat{\to}nat) & \text{use Function Inclusion Law} \\
= & (nat\colon nat)\ \wedge\ \wedge\langle n\colon nat \to suc\ n\colon nat\rangle & \\
= & \wedge\langle n\colon nat \to n+1\colon nat\rangle & \\
= & \top &
\end{array}
$$

And, more generally,

$$(f\colon A{\to}B) \quad = \quad (A\colon \mathcal{D}f)\ \wedge\ \wedge\langle a\colon A\cdot fa\colon B\rangle$$

We can similarly show

$$\langle d\colon nat+1 \to \langle n\colon nat \to n\colon d{\times}nat\rangle\rangle\colon\ (nat+1){\to}nat{\to}bin$$

The function eat defined earlier with a variable number of operands satisfies

$$eat\colon nat \to (0, eat)$$

The use of bunches unified our treatment of numbers and number types; it similarly unifies our treatment of functions and function types.

Let *check* be a function whose variable is a function.

$$check \;=\; \langle f\colon ((0,..10)\to int) \to \wedge\langle n\colon (0,..10) \to even(fn)\rangle\rangle$$
$$even \;=\; \langle i\colon int \to i\colon 2\times int\rangle$$

Function *check* checks whether a function, when applied to the first 10 natural numbers, produces only even integers. An operand for *check* must be a function whose domain includes 0,..10 because *check* will be applying its operand to all elements in 0,..10 . An operand for *check* must be a function whose results, when applied to the first 10 natural numbers, are included in *int* because *even* will be applied to them. An operand for *check* may have a larger domain (extra domain elements will be ignored), and it may have a smaller range. If $A\colon B$ and $f\colon B\to C$ and $C\colon D$ then $f\colon A\to D$. Therefore

$$suc\colon (0,..10)\to int$$

We can apply *check* to *suc* and the result will be $\bot$. (We are applying a function to a non-element operand, so we ought to decide whether *check* distributes over its operand. But, thanks to the Function Inclusion Laws, the result is the same either way.)

Function Composition

Let f and g be functions such that $-(f\colon \mathcal{D}g)$ (f is not in the domain of g). Then $g\,f$ is the composition of f and g , defined by the Composition Laws

$$\mathcal{D}(g\,f) \;=\; \S\langle x\colon \mathcal{D}f \to fx\colon \mathcal{D}g\rangle$$
$$(g\,f)\,x \;=\; g\,(f\,x)$$

Because composition is associative,

$$f\,(g\,h) \;=\; (f\,g)\,h$$

we don't need the parentheses.

The Composition Laws let us write complicated combinations of functions and operands without parentheses. They sort themselves out properly according to their domains. For example, suppose f and g are functions of one variable, and h is a function of two variables. Suppose further $-(f\colon \mathcal{D}h)$ (f is not in the domain of h), and $-(g\colon \mathcal{D}(h(fx)))$ (g is not in the domain of $h(fx)$). Then

$$\begin{array}{lll}
 & h\,f\,x\,g\,y & \text{juxtaposition is left-to-right}\\
= & (((h\,f)\,x)\,g)\,y & \text{use function composition on } h\,f \text{ since } -(f\colon \mathcal{D}h)\\
= & ((h\,(f\,x))\,g)\,y & \text{use function composition on } (h\,(f\,x))\,g \text{ since } -(g\colon \mathcal{D}(h(fx)))\\
= & (h\,(f\,x))\,(g\,y) & \text{drop superfluous parentheses}\\
= & h\,(f\,x)\,(g\,y) &
\end{array}$$

Operator-Function Composition

If $x\colon D$, then application yields

$$\langle y\colon D \to -y\rangle\,x \;=\; -x$$

so in the context of application, the function $\langle y\colon D \to -y\rangle$ is identical to the operator $-$ on domain D . We take them to be identical also in the context of composition. If $-(f\colon D)$, then

$$\begin{array}{ll}
 & (-f)\,x\\
= & (\langle y\colon D \to -y\rangle\,f)\,x\\
= & \langle y\colon D \to -y\rangle\,(f\,x)\\
= & -(f\,x)
\end{array}$$

The $-$ operator is being composed with a function. We can similarly compose any operator with a function if the operator does not operate on the function but on its result. For example, if h is a function whose result is a function whose result is a number, then

$(\times h)\ x\ =\ \times(h\ x)$
$\times\langle x \rightarrow hx\rangle\ =\ \langle x \rightarrow \times(hx)\rangle$
Since the operator # does apply to a function, it cannot be composed with a function.

Infix operators that do not operate on functions but on their results can similarly be composed with the functions. For example, if f and g are functions whose results are numbers a and b,
$(f + g)\ x\ =\ f\,x + g\,x$
$\langle x \rightarrow fx\rangle + \langle x \rightarrow gx\rangle\ =\ \langle x \rightarrow fx{+}gx\rangle$
So the inner product of f and g is $\times(f{+}g)$.

Since the infix operators = and : do operate on functions, they cannot be composed with functions. (Operator-function composition has been called "lifting".)

Aside The following alternative was considered and rejected. It would be more uniform to say that all operators compose with functions, so that
$(f + g)\ x\ =\ (f\,x + g\,x)$
$(f = g)\ x\ =\ (f\,x = g\,x)$
$(f : g)\ x\ =\ (f\,x : g\,x)$
and then, for functions f and g having the same domain

$\times(f{+}g)\ =\ \times\langle x \rightarrow fx{+}gx\rangle$	inner product
$\wedge(f{=}g)\ =\ \wedge\langle x \rightarrow fx{=}gx\rangle$	function equality
$\wedge(f{:}g)\ =\ \wedge\langle x \rightarrow fx{:}gx\rangle$	function inclusion

and so on. *End of Aside*

Selective Union

If f and g are functions, then
$f\,|\,g$ "f otherwise g"
is a function that behaves like f when applied to an operand in the domain of f, and otherwise behaves like g. The laws are:
$\mathcal{D}(f\,|\,g)\ =\ \mathcal{D}f, \mathcal{D}g$
$(f\,|\,g)\ \mathrm{x}\ =\ fx \triangleleft x{:}\ \mathcal{D}f \triangleright gx$

Selective union is idempotent, associative, and composition distributes over it.
$f\,|\,f\ = f$
$f\,|\,(g\,|\,h)\ =\ (f\,|\,g)\,|\,h$
$(g\,|\,h)\,f\ =\ g\,f\,|\,h\,f$

Selective union gives us a way to express a function by listing domain-range pairs as in the following example:
$0 \rightarrow 2\ |\ 2 \rightarrow 1\ |\ 1 \rightarrow 0$
When the domain values are textual, we have the familiar "record" or "structure" from various programming languages; by letting the range values be bunches we have "record types".

Limits

Let f be a function with domain *nat* (an infinitely long list). Function application determines the result of applying f to any element of *nat*. We now say what happens when we apply f to $\top$. Define $f\top$ by the following Limit Law:

$$\vee\langle m \to \wedge\langle n \to f(m+n)\rangle\rangle \;\le\; f\top \;\le\; \wedge\langle m \to \vee\langle n \to f(m+n)\rangle\rangle$$

with all domains being *nat* .

For some functions f , the Limit Law tells us $f\top$ exactly. For example,

$$\langle n\text{: } nat \to 1/(n+1)\rangle\top \;=\; 0$$

For nondecreasing f , $f\top = \vee f$. For nonincreasing f , $f\top = \wedge f$. For example,

$$\wedge\langle n\text{: } nat \to 1/(n+1)\rangle \;=\; 0$$

For some functions, the Limit Law tells us a little less. For example,

$$-1 \le \langle n\text{: } nat \to (-1)^n\rangle\top \;\le\; 1$$

In general,

$$\wedge f \le f\top \le \vee f$$

Now that we have defined $f\top$ where f is any function with domain *nat* , we can define the real numbers. First we define *xreal* as follows:

$$xreal \;=\; (nat{\to}rat)\top$$

Notice that $nat{\to}rat$ includes all functions with domain *nat* and range *rat* , and we take the limits of all such functions. And now

$$real \;=\; §\langle x\text{: } xreal \to \bot < x < \top\rangle$$

Regrets

A list L is very similar to the function $\langle n\text{: } (0,..\#L) \to Ln\rangle$. Indexing a list is the same as function application, and the same notation $L\ n$ is used. List composition is the same as function composition, and the same notation $L\ M$ is used. List length is the same as function size, and the same notation is used. It is useful to mix lists and other functions in a composition. For example,

$$suc\ [3;\ 5;\ 2] \;=\; [4;\ 6;\ 3]$$

We can also mix lists and functions in a selective union. With function $1{\to}21$ as left operand, and list [10; 11; 12] as right operand, we get

$$1{\to}21 \mid [10;\ 11;\ 12] \;=\; [10;\ 21;\ 12]$$

just as we defined it for lists. And $+L$ conveniently expresses the sum of the list. So I really want to unify lists and functions.

Unfortunately, that would mean $[S;\ T]\text{: } [S]$. Texts are lists of characters, and that law says that “ab”: “a” . So

$$(\text{“a”} \to 0 \mid \text{“ab”} \to 1)\ \text{“ab”} \;=\; 0$$

This is not how we want a record structure to work.

Due to the Function Inclusion Law, there can be no way to express “all functions of reals”. If we were to unify lists and functions, there would be no way to express “all lists of reals”, nor “all lists of X ” for any X . For suppose *ALN* expresses all lists of naturals. Then

$$[3;\ -3]\text{: } [3]\text{: } ALN$$

Another nice unification would be bunches with functions whose result is binary. In this unification, bunch inclusion would be “backwards application”.

$$(x\text{: } A) = A\ x$$

And there would be no reason to keep both notations.

Unresolved

The following function would appear to apply to functions of reals and yield the average:

$$\langle f\text{: } ((0,..\#f) \to \mathit{real}) \to +f/\#f\rangle$$

What makes this example interesting is that the variable f introduced in the function is used in the domain $((0,..\#f)\to \mathit{real})$ of the variable. A function introduces a variable, in this example f, and a law, in this case $f\text{: } ((0,..\#f)\to \mathit{real})$, locally. There is no reason to prevent a variable from occurring twice in a law. But if the domain mentions the variable, there's no way to determine what the domain is, and so no way to apply the function. Perhaps the Domain Law should be

$$(x\text{: } \mathcal{D}\langle x\text{: } Dx \to Rx\rangle) \quad = \quad (x\text{: } Dx)$$

But then $\mathcal{D}\langle x\text{: } x \to 0\rangle$ might give Russell's paradox.

Probability

The standard theory of probability assigns 0 to an event that cannot happen, $1/2$ to an event that is equally likely to happen or not happen, and 1 to an event that is certain to happen. In a set of events in which exactly one event must happen, the probabilities sum to 1. The integral of a probability distribution must be 1.

Perhaps there is another way to develop probablility theory based on unified algebra. Perhaps an event that cannot happen has probability $\bot$, an event that is equally likely to happen or not happen has probability 0, and an event that is certain to happen has probability $\top$. In a set of events in which exactly one event must happen, the average probability is 0. The integral of a probability distribution must be 0. In standard probability theory, if there are n equally likely events, they each have probability $1/n$, for finite n. But there is no way to say that an infinite number of events are equally probable. In this proposed new theory, n equally likely events each have probability 0, even if n is infinite.

Perhaps the new probability space is related to the logarithm of the old space; essentially, probabilities are replaced by information content. The hope is that the complicated formulas for distributions in the standard theory can be simplified by transforming the space of probabilities.

Conclusion

We have presented an algebra that unifies numbers with booleans, types with values, function spaces with functions. There is no loss of structure, just loss of duplication. This is mathematics by design. Like any design, it is neither right nor wrong; the criteria for judging it are usefulness and elegance.

When we apply a formalism to describe and reason about some phenomena, we may find that it works quite well for a certain range of observations, but less well outside that range. In this formalism, when D represents a finite class of objects and Bx is a binary expression, we find that $\vee\langle x\text{: } D \to Bx\rangle$ is quite useful for saying “There exists an object, let's call it x, in the class of objects represented by D, such that B is true of x.”. But when D represents an infinite class of objects, $\vee\langle x\text{: } D \to Bx\rangle$ differs from the traditional mathematical idea of existence. One way to resolve the discrepancy is to redesign the algebra, sacrificing simplicity and elegance, to attempt to fit the traditional mathematical idea of existence. Another way to resolve the discrepancy is to change the traditional mathematical idea of existence to fit the algebra, or even abolish the idea of mathematical existence altogether. I recommend the latter.

Acknowledgments

I thank Rutger Dijkstra, Wim Hesselink, Jim Grundy and David Barton for catching some errors. Remaining errors are, of course, for the purpose of testing the reader. I thank Dimitrie Paun for discussion.

References

[0] J.Grundy: “Transformational Hierarchical Reasoning”, *the Computer Journal* 39(4) 291-302, 1996 May

[1] E.C.R.Hehner: “from Boolean Algebra to Unified Algebra”, www.cs.toronto.edu/~hehner/BAUA.pdf

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

[3] C.A.R.Hoare: “a couple of novelties in the propositional calculus”, *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik* 31(2), 173-8, 1985