Unified Algebra

Eric C. R. Hehner
University of Toronto

Introduction

Mathematics has evolved. Bits of it are created by many people over a long 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 just the parts that are useful, unifying the parts that are similar.

I present a unified algebra that includes what are commonly called boolean algebra, number algebra, sets, lists, functions, quantification, type theory, and limits; this mathematics forms the foundation for much of computer science. 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 algebra, I rely on the reader's experience in using algebra. For motivations, justifications, and commentary, I refer the reader to [0].

The algebra is presented from the very beginning, leaving out nothing. That makes the early parts of the presentation very basic, but readers may appreciate the care and effort required to design a simple and general algebra. And it's a nontrivial problem to get the presentation started without ever saying “trust me for now, I'll make this clear later”. Anyone interested in implementation on a computer must pay attention to micro-mechanical detail. The viewpoint I adopt throughout is formalist, as required for implementation.

I begin with boolean algebra, renamed “binary algebra”, and its two extremes, renamed “top” and “bottom”. That's the only new terminology. By contrast, 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, 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

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

Expressions and Values

An algebra consists of expressions, which are used to express values in an application. For example, the values may be amounts of water, or voltage, or frequency of vibration, or guilt and innocence. Here are four definitions that precede all choice of symbols and rules of any algebra.

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

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.) Consistency is essential; completeness is not. Expressiveness is desirable; uniqueness is not. In general, several expressions may represent the same value. When we say that $2+3$ is 5 , we do not mean that $2+3$ and 5 are the same expression; clearly they are not. We mean that the value represented by $2+3$ is the value represented by 5 . When we say that $2+3$ has value 5 , we again mean that expression $2+3$ represents the same value that expression 5 represents. We might just as well say that 5 has value $2+3$.

Expression Structure

An expression can be a part of a larger expression, in which case it is called a “subexpression” of the larger expression. One way to make a larger expression from a subexpression is to write a symbol, such as – , followed by the subexpression. The symbol is called an “operator”, and the subexpression is called its “operand”. Another way to make a larger expression is to write two subexpressions with a symbol, such as + , between them.

Placing operators between operands makes the structure of 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	
	if then else fi within which the order of evaluation again applies	
	subscript x_n superscript x^n	right to left
1	juxtaposition $f\,x$	left to right
2	one operand – ¢ $ ~ ⁓ ϟ □ ↔ # → ∧ ∨ = ⧧ § + × ⩓ ⩔ ◊	right to left
	two operands ⟶	right to left
3	two operands × / ∧ ∨ △ ▽	left to right
4	two operands + – ⁺	left to right
5	two operands , ,.. ‘ ; ;.. |	
6	three operands ◃ ▹ precedence applies to first and last operands	
7	two operands = ⧧ < > ≤ ≥ : ∊	

In the order of evaluation, two-operand + can be found on level 4, and two-operand × on level 3; that means, in the absence of parentheses, evaluate two-operand × before two-operand + . The example $2+3\times4$ therefore means the same as $2+(3\times4)$. Within levels 1, 3, and 4 evaluation is from left to right. Within level 2 evaluation is from right to left. On level 7, $x=y=z$ means the same as $(x=y)\wedge(y=z)$, and similarly for the other operators and mixtures of operators on that level.

Format

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

Constants, Variables, and Instantiation

Constants and variables are kinds of expressions. In this paper we use single italic letters like x for variables (but that's not a principle of unified algebra), and a variety of notations for constants. Expressions represent values; a constant represents a particular value, and a variable represents an arbitrary value. A variable can be replaced by another expression; this is called "instantiation". A constant cannot be replaced. Expression -2 is called an "instance" of expression $-x$. Here is how instantiation works.

- 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. Both $2+2$ and $2+3$ are instances of $x+y$.

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

Evaluation Rules

Here are the rules to determine the value of expressions. The rules are independent of the choice of expressions and values. (The examples use symbols, but the rules do not.)

<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 $\top{\wedge}\top$ is $\top$.

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

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, north and south poles of magnets. 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 ”
$x\ne y$	“ x differ y ”
$x<y$	“ x below y ”
$x>y$	“ x above y ”
$x\le y$	“ x at most y ”
$x\ge y$	“ x at least y ”
$x\wedge y$	“ x min y ” (arms down)
$x\vee y$	“ x max y ” (arms up)
$x\triangle y$	“ x neg min y ”
$x\triangledown y$	“ x neg max y ”
if x **then** y **else** z **fi**	“if x then y else z fee”

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

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

xy	$\top\top$	$\top\bot$	$\bot\top$	$\bot\bot$
$x=y$	$\top$	$\bot$	$\bot$	$\top$
$x \neq y$	$\bot$	$\top$	$\top$	$\bot$
$x<y$	$\bot$	$\bot$	$\top$	$\bot$
$x>y$	$\bot$	$\top$	$\bot$	$\bot$
$x \leq y$	$\top$	$\bot$	$\top$	$\top$
$x \geq y$	$\top$	$\top$	$\bot$	$\top$
$x \wedge y$	$\top$	$\bot$	$\bot$	$\bot$
$x \vee y$	$\top$	$\top$	$\top$	$\bot$
$x \triangle y$	$\bot$	$\top$	$\top$	$\top$
$x \triangledown y$	$\bot$	$\bot$	$\bot$	$\top$

xyz	$\top\top\top$	$\top\top\bot$	$\top\bot\top$	$\top\bot\bot$	$\bot\top\top$	$\bot\top\bot$	$\bot\bot\top$	$\bot\bot\bot$
if x **then** y **else** z **fi**	$\top$	$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\top$	$\bot$

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 value of $\top$ in four ways. One way is to abbreviate the statement "Expression x has the same value as $\top$." by just writing x , without saying anything about it. Whenever we just write a binary expression, we mean that it has the same value as $\top$ (expresses the same value that $\top$ expresses). For example, instead of saying "Expression $\top=\top$ has the same value as $\top$." we just say " $\top=\top$ " ("top equals top"; remember that truth is just one of the binary values in just one of the applications).

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 value of $\top$; we have no name for an assignment that gives it the value of $\bot$. A law is a binary expression for which any assignment of values to its variables gives it the value $\top$, and so by the Completion Rule it too has the value $\top$; we have no name for a binary expression that has the value $\bot$.

We often use the Indirect Rule by stating that an expression is a law, which means we are assigning it the same value as $\top$. If we want to assign $\bot$'s value to expression x , instead we state the law $-x$, and then rely on the Consistency Rule to say that x is $\bot$. 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 the same value as $\top$, or more briefly, we say $x=y$.

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

Ternary Algebra

Between the two values represented by $\top$ and $\bot$, we now consider another 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 expressions, we extend the value tables.

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

xy	$\top\top$	$\top 0$	$\top\bot$	$0\top$	00	$0\bot$	$\bot\top$	$\bot 0$	$\bot\bot$
$x=y$	$\top$	$\bot$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\bot$	$\top$
$x\neq y$	$\bot$	$\top$	$\top$	$\top$	$\bot$	$\top$	$\top$	$\top$	$\bot$
$x<y$	$\bot$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\top$	$\top$	$\bot$
$x>y$	$\bot$	$\top$	$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\bot$
$x\le y$	$\top$	$\bot$	$\bot$	$\top$	$\top$	$\bot$	$\top$	$\top$	$\top$
$x\ge y$	$\top$	$\top$	$\top$	$\bot$	$\top$	$\top$	$\bot$	$\bot$	$\top$
$x\wedge y$	$\top$	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 $x=-x$ has no solution in binary algebra because both assignments of binary values give it the value $\bot$; in ternary algebra it has solution 0 . 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 is 0 , $x\vee-x$ is 0 . By extending the algebra, we have gained some solutions and lost some laws.

We can add many new ternary expressions. For example, we can add approximate equality and modular (circular) addition 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	$\bot$	$\top$

Four and More Values

To design a four-valued algebra as a consistent extension of ternary algebra, we add a new value represented by θ . Like 0 , θ is equal to its own negation, and is situated between $\top$ and $\bot$, but 0 and θ are unrelated in the ordering. There are two other ways to design a four-valued algebra as a consistent extension of binary algebra. And there are many interesting algebras with more values. We now leap to an infinite-valued totally-ordered algebra. Value tables, which are already cumbersome for ternary algebra (**if** x **then** y **else** z **fi** takes 27 columns), become impossible with infinitely many values, so from now on we give values to new expressions by stating laws.

Common Laws

I have introduced binary and ternary expressions, and mentioned expressions of four or more values. I am about to introduce numbers, bunches, sets, strings, lists, and functions by saying how to write them and giving their laws. There are some laws of binary algebra that are not laws of any other algebra; for example,

$(x \le y) = -x \lor y$	material order
$((x=y)=z) = (x=(y=z))$	associative
$((x ⧧ y) ⧧ z) = (x ⧧ (y ⧧ z))$	associative
$(x=\top) = x$	identity
$(x ⧧ \bot) = x$	identity

The next two expressions are laws of binary algebra, and one of the four-valued algebras mentioned in the previous section, but not of any other algebra mentioned in this paper.

$x \lor -x$	excluded middle
$-(x \land -x)$	noncontradiction

There are laws of some of our algebras that are not laws of binary algebra, but only because they employ symbols that are not symbols of binary algebra. Any law of any of our algebras that employs only the symbols of binary algebra is also a law of binary algebra.

There are many laws that are common to all of the algebras in this paper; for example,

$\bot \le x \le \top$	extremes
$x \land \bot = \bot$	base
$x \lor \top = \top$	base
$x \vartriangle \bot = \top$	base
$x \triangledown \top = \bot$	base
$x \land \top = x$	identity
$x \lor \bot = x$	identity
$x = x$	reflexivity
$x \le x$	reflexivity
$x \ge x$	reflexivity
$-(x<x)$	irreflexivity
$-(x>x)$	irreflexivity
$- -x = x$	double negation or self-inverse
$x \land x = x$	idempotence
$x \lor x = x$	idempotence
$(x=y) = (y=x)$	symmetry
$(x ⧧ y) = (y ⧧ x)$	symmetry
$x \land y = y \land x$	symmetry
$x \lor y = y \lor x$	symmetry

$x \triangle y \;=\; y \triangle x$	symmetry
$x \triangledown y \;=\; y \triangledown x$	symmetry
$-(x < y < x)$	antisymmetry
$-(x > y > x)$	antisymmetry
$-(x < y = x)$	exclusivity
$-(x > y = x)$	exclusivity
$(x \le y) \;=\; (x<y) \vee (x=y)$	inclusivity
$(x \ge y) \;=\; (x>y) \vee (x=y)$	inclusivity
$(x>y) \;=\; (y<x)$	mirror
$(x \ge y) \;=\; (y \le x)$	mirror
$(x<y) = (-x > -y)$	reflection
$(x \wedge y = x) = (x \le y) = (y = x \vee y)$	connection
$x \wedge (x \vee y) = x$	absorption
$x \vee (x \wedge y) = x$	absorption
$-(x=y) \;=\; (-x \ne -y)$	duality
$-(x \ne y) \;=\; (-x = -y)$	duality
$-(x<y) \;=\; (-x \le -y)$	duality
$-(x \le y) \;=\; (-x < -y)$	duality
$-(x>y) \;=\; (-x \ge -y)$	duality
$-(x \ge y) \;=\; (-x > -y)$	duality
$-(x \wedge y) \;=\; -x \vee -y$	duality
$-(x \vee y) \;=\; -x \wedge -y$	duality
$-(x \triangle y) \;=\; -x \triangledown -y$	duality
$-(x \triangledown y) \;=\; -x \triangle -y$	duality
$x \wedge (x \le y) \;\le\; y$	modus ponens
$(x \ne y) = -(x=y)$	unequality
$x \triangle y \;=\; -(x \wedge y)$	neg min
$x \triangledown y \;=\; -(x \vee y)$	neg max
$(x \wedge y) \wedge z \;=\; x \wedge (y \wedge z)$	associativity
$(x \vee y) \vee z \;=\; x \vee (y \vee z)$	associativity
$(x = y = z) \;\le\; (x=z)$	transitivity
$(x < y < z) \;\le\; (x<z)$	transitivity
$(x > y > z) \;\le\; (x>z)$	transitivity
$(x \le y \le z) \;\le\; (x \le z)$	transitivity
$(x \ge y \ge z) \;\le\; (x \ge z)$	transitivity
$x \wedge y \;\le\; y \;\le\; y \vee z$	specialization and generalization
$x \wedge (y \vee z) \;=\; (x \wedge y) \vee (x \wedge z)$	distribution or factoring
$x \vee (y \wedge z) \;=\; (x \vee y) \wedge (x \vee z)$	distribution or factoring
$(x \le y \wedge z) \;=\; (x \le y) \wedge (x \le z)$	distribution or factoring
$(x \le y \vee z) \;\ge\; (x \le y) \vee (x \le z)$	distribution or factoring
$(x \wedge y \le z) \;\ge\; (x \le z) \vee (y \le z)$	antidistribution
$(x \vee y \le z) \;=\; (x \le z) \wedge (y \le z)$	antidistribution
$(w \wedge x) \vee (y \wedge z) \;\le\; (w \vee y) \wedge (x \vee z)$	
$\mathbf{if}\ \top\ \mathbf{then}\ x\ \mathbf{else}\ y\ \mathbf{fi} \;=\; x$	base
$\mathbf{if}\ \bot\ \mathbf{then}\ x\ \mathbf{else}\ y\ \mathbf{fi} \;=\; y$	base
$-\ \mathbf{if}\ x\ \mathbf{then}\ y\ \mathbf{else}\ z\ \mathbf{fi} \;=\; \mathbf{if}\ x\ \mathbf{then}\ -y\ \mathbf{else}\ -z\ \mathbf{fi}$	distribution or factoring

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

I now introduce infinitely many values between $\top$ and $\bot$. Here are some of them.

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

All operators apply to all values.

The expressions of number algebra are called “number expressions”. They can be used to represent anything that comes in 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 $5296\wedge375$ 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 or factoring
$x + y\vee z = (x+y) \vee (x+z)$	distributivity or factoring
$x + y\triangle z = (x-y) \vee (x-z)$	
$x + y\triangledown z = (x-y) \wedge (x-z)$	
w + **if** x **then** y **else** z **fi** = **if** x **then** $w+y$ **else** $w+z$ **fi**	distributivity or factoring
$-x = 0 - x$	negation
$-(x+y) = -x + -y$	distributivity or factoring
$-(x-y) = -x - -y$	distributivity or factoring
$-(x\times y) = (-x)\times y$	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 or factoring
$x\times(y\times z) = (x\times y)\times z$	associativity
$(\bot<x<\top) \land (x\neq 0) \le ((x\times y = x\times z) = (y=z))$	cancellation
$(0<x) \le (x\times\top = \top)$	absorption
$(0<x) \le (x\times\bot = \bot)$	absorption
$x/1 = x$	identity
$(\bot<x<\top) \land (x\neq 0) \le (x/x = 1)$	inverse
$x\times(y/z) = (x\times y)/z = x/(z/y)$	multiplication-division
$(y\neq 0) \le (x/(y/z) = x/(y\times z))$	multiplication-division
$(\bot<x<\top) \le (x/\top = 0 = x/\bot)$	annihilation
$(\bot<x<\top) \le (x^0 = 1)$	base
$x^1 = x$	identity
$x^{y+z} = x^y\times x^z$	exponents
$x^{y\times z} = (x^y)^z$	exponents
$\bot<0<1<\top$	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
$(x<y) \lor (x=y) \lor (x>y)$	trichotomy

Calculation

Given an expression, we might find a simpler expression with the same value. For example,

$$\begin{array}{lll}
 & x\times(z+1) - y\times(z-1) - z\times(x-y) & \text{distribute} \\
= & (x\times z + x\times 1) - (y\times z - y\times 1) - (z\times x - z\times y) & \text{unity and double negation} \\
= & x\times z + x - y\times z + y - z\times x + z\times y & \text{symmetry and associativity} \\
= & x + y + (x\times z - x\times z) + (y\times z - y\times z) & \text{zero and identity} \\
= & x+y &
\end{array}$$

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

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

By simply writing it, we are saying that it has the same value as $\top$. The hint "distribute" is intended to make it clear that

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

is $\top$; the hint "unity and double negation" is intended to make it clear that

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

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

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

is $\top$, 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,

	$x\times(x + 2)$	distribute
$=$	$x^2 + 2\times x$	identity and zero
$=$	$x^2 + 2\times x + 1 - 1$	factor
$=$	$(x+1)^2 - 1$	a square is nonnegative
$\ge$	-1	

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

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. If the calculation is input to an automated tool, few hints are needed, and they must be expressed formally, but we do not pursue that here.

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.

	$a \triangle (a \bigtriangledown b)$	use neg min and neg max laws
$=$	$-(a \wedge -(a\vee b))$	decrease $a\vee b$ to a and so decrease the whole expression
$\ge$	$-(a \wedge -a)$	use a previous example
$=$	$-\bot$	
$=$	$\top$	

And so $a \triangle (a\bigtriangledown b)$ is above or equal to $\top$, and since there is nothing above $\top$, it 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 .
$x\triangle y$ varies inversely with x and inversely with y .
$x\bigtriangledown y$ 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 .
if x **then** y **else** z **fi** varies directly with y 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.

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

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

In $x<y$,	when simplifying x , we can assume y is not $\bot$;
	when simplifying y , we can assume x is not $\top$.
In $x>y$,	when simplifying x , we can assume y is not $\top$;
	when simplifying y , we can assume x is not $\bot$.
In $x\le y$,	when simplifying x , we can assume y is not $\top$;
	when simplifying y , we can assume x is not $\bot$.
In $x\ge y$,	when simplifying x , we can assume y is not $\bot$;
	when simplifying y , we can assume x is not $\top$.
In $x\land y$,	when simplifying x , we can assume y is not $\bot$;
	when simplifying y , we can assume x is not $\bot$.
In $x\lor y$,	when simplifying x , we can assume y is not $\top$;
	when simplifying y , we can assume x is not $\top$.
In $x\vartriangle y$,	when simplifying x , we can assume y is not $\bot$;
	when simplifying y , we can assume x is not $\bot$.
In $x\triangledown y$,	when simplifying x , we can assume y is not $\top$;
	when simplifying y , we can assume x is not $\top$.
In **if** x **then** y **else** z **fi** ,	when simplifying y , we can assume x is not $\bot$;
	when simplifying z , we can assume x is not $\top$.

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) 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 $0, 2, 5, 9$.

I earlier made the statement "We must never use an expression to express more than one value; to do so would be a serious error called inconsistency.". I now amend that statement to say "We must never use an elementary expression to express more than one value.". Bunch expressions can indeed represent more than one value; that is their purpose, and we do not call it "inconsistency".

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 [1].

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

null	empty	
bin	the binaries	includes $\top, \bot$
nat	the naturals	includes $0, 1, 2$ and others
int	the integers	includes $-2, -1, 0, 1, 2$ and others
rat	the rationals	includes $-1, 0, 2/3$ and others
real	the reals	

We define them formally in a moment.

The operators $, \text{'}\ ¢\ :\ =\ \ddagger$ **if then else fi** apply to bunch operands according to the axioms already presented. Other operators can be applied to bunches by applying them to the elements of the bunch. For example,

$-null = null$	base
$-(A, B) = -A, -B$	distribution or factoring
$A+null = null$	base
$A+(B, C) = A+B, A+C$	distribution or factoring

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

$$null: A$$
$$(¢A = 0) \quad = \quad (A = null)$$

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

The bunch *nat* is defined by two laws.

$0, nat+1: nat$	construction
$(0, B+1: B) \ \le\ (nat: B)$	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 is integer, and n is integer or binary, and $m\le n$. This notation means the bunch $m, m+1, m+2$, up to but not including n . The asymmetric notation is a reminder that the left end is included but the right end is excluded. Here are its laws:

$$(x: m,..n) \quad = \quad (x: int) \wedge (m\le x<n)$$
$$¢(m,..n) = n-m$$

And here are three examples:

$$0,..3 = 0, 1, 2$$
$$0,..0 = null$$
$$0,..\top = nat$$

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 A is any set, then

$\sim A$	"contents of A "

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*	set formation
~{*A*} = *A*	contents
{*A*} ⧧ *A*	structure
${*A*} = ¢*A*	size
(*A* ∊ {*B*}) = (*A*: *B*)	element
({*A*} ≤ {*B*}) = (*A*: *B*)	subset
({*A*}: ϟ*B*) = (*A*: *B*)	power
{*A*} ∨ {*B*} = {*A*, *B*}	union
{*A*} ∧ {*B*} = {*A* ‘ *B*}	intersection
({*A*} = {*B*}) = (*A* = *B*)	equation

Note that the element, subset, and power 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 $\leftrightarrow$ operator.

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

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.

If i is an item, S and T are strings, and n is a natural number, 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 ”
$S\triangleleft n\triangleright i$	“ S but at n there's i ”

are strings, and

$\leftrightarrow S$	“length of S ”

is a natural number or $\top$, and

$S=T$	“ S equals T ”
$S\ne T$	“ 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.

$nil;\ S = S;\ nil = S$	$S_{null} = null$
$S;\ (T;\ U) = (S;\ T);\ U$	$S_{T,\,U} = S_T\, ,\, S_U$
$\leftrightarrow nil = 0$	$S_{\{T\}} = \{S_T\}$
$\leftrightarrow i = 1$	$S_{nil} = nil$
$\leftrightarrow(S;\ T) = \leftrightarrow S + \leftrightarrow T$	$S_{T;\,U} = S_T\, ;\, S_U$
$(\leftrightarrow S<\top) \le ((S;\ i;\ T)_{\leftrightarrow S} = i)$	$S_{(T_U)} = (S_T)_U$
$(\leftrightarrow S<\top) \le ((i<j) \le (S;\ i;\ T < S;\ j;\ U))$	$(\leftrightarrow S<\top) \le (nil \le S < S;\ i;\ T)$
$(\leftrightarrow S<\top) \le ((i=j) = (S;\ i;\ T = S;\ j;\ T))$	$(\leftrightarrow S<\top) \le (S;i;T \triangleleft \leftrightarrow S \triangleright j = S;j;T)$

We also use the notation

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

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

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

Here are the laws.

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

String catenation distributes over bunch union:

$A;\ null;\ B = null$	base
$(A, B);\ (C, D) = (A;C), (A;D), (B;C), (B;D)$	distribution or factoring

So a string of bunches is equal to a bunch of strings. Thus, for example,

$0;\ 1;\ 2\colon\ \ nat;\ 1;\ (0,..10)$

because $0\colon nat$ and $1\colon 1$ and $2\colon 0,..10$.

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.

$[null] = null$	base
$[A, B] = [A], [B]$	distribution or factoring

Because of the distribution we can say

$[0;\ 1;\ 2]\colon\ \ [nat;\ 1;\ (0,..10)]$

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.

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,

$\sim L$	"contents of L "

is a string,

$\#L$	"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 contents of a list is the string of items it contains.

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

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

$$2\rightarrow 22 \mid [10;..15] \ = \ [10;\ 11;\ 22;\ 13;\ 14]$$
$$2\rightarrow 22 \mid 3\rightarrow 33 \mid [10;..15] \ = \ [10;\ 11;\ 22;\ 33;\ 14]$$

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

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

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, let i and j be items, and let n be a natural number.

$[S] \ \neq \ S \ = \ \sim[S] \ = \ [\sim S]$	structure, contents, and formation
$\#[S] \ = \ \leftrightarrow S$	length
$[S]{}^{+}[T] \ = \ [S;\ T]$	catenation
$[S]\ T \ = \ S_T$	indexing
$[S_T] \ = \ S_{[T]}$	
$[S]\ [T] \ = \ [S_T]$	composition
$n \rightarrow i \mid [S] \ = \ [S\triangleleft n\triangleright i]$	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 or factoring

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

$$\begin{aligned} A \ = \ [\ & [6;\ 3;\ 7;\ 0]\ ; \\ & [4;\ 9;\ 2;\ 5]\ ; \\ & [1;\ 5;\ 8;\ 3]\ ] \end{aligned}$$

Then A is a 2-dimensional array, or more particularly, a $3{\times}4$ 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:

$\langle \textit{variable}\colon \textit{domain} \rightarrow \textit{result}\rangle$

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. The local variable is an element, and *variable*: *domain* is a local law within the function. For example, the successor function

$\langle n\colon nat \rightarrow n{+}1\rangle$

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

$\langle n\colon nat \rightarrow 1\rangle$

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, if the domain is known to be *nat* , the preceding constant function can be written

${\rightarrow}1$

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

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

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.

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

The constant function of two natural variables

$\langle n\colon nat \rightarrow \langle m\colon nat \rightarrow 0\rangle\rangle$

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. For example, in

$\langle x\colon nat \rightarrow x{+}y\rangle$

variable x is local, and variable y is nonlocal. 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

$f\,x$, pronounced "f applied to x" or "f of x". For example, since 3 is an element of nat ,

$$\langle x\colon nat \rightarrow x{+}y\rangle\ 3$$

is a function application, and it expresses (has the same value as) the instantiation that replaces x in $x{+}y$ with 3 .

$$\langle x\colon nat \rightarrow x{+}y\rangle\ 3 \ = \ 3{+}y$$

Here is another example.

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

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

$$eat \ = \ \langle n\colon nat \rightarrow \mathbf{if}\ n{=}0\ \mathbf{then}\ 0\ \mathbf{else}\ eat\ \mathbf{fi}\rangle$$

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

Here is the Application Law. If x is an element of D , then

$$\langle x\colon D \rightarrow R\rangle\ x \ = \ R$$

Here is an instance of this law, replacing D with nat and R with $x{+}y$.

$$\langle x\colon nat \rightarrow x{+}y\rangle\ x \ = \ x{+}y$$

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 $\langle x\colon nat \rightarrow x{+}y\rangle\ x$ by replacing x with y we obtain $\langle x\colon nat \rightarrow x{+}y\rangle\ y$.

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

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

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

$$\square\langle x\colon D \rightarrow Rx\rangle = D$$

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:

$$\langle x\colon \square f \rightarrow fx\rangle = f$$

This law can be instantiated by replacing f with $\langle y\colon D \rightarrow fy\rangle$ to obtain

$$\langle x\colon \square\langle y\colon D \rightarrow fy\rangle \rightarrow \langle y\colon D \rightarrow fy\rangle\ x\rangle = \langle y\colon D \rightarrow fy\rangle$$

We use the Domain Law, and if x is an element in D then we can apply the middle $\langle y\colon D \rightarrow fy\rangle$ to x to obtain

$$\langle x\colon D \rightarrow fx\rangle = \langle y\colon D \rightarrow fy\rangle$$

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.

$$\#f \ = \ \cent\square f$$

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

$$\mathbf{if}\ b\ \mathbf{then}\ f\ \mathbf{else}\ g\ \mathbf{fi}\ x \ = \ \mathbf{if}\ b\ \mathbf{then}\ f\,x\ \mathbf{else}\ g\,x\ \mathbf{fi}$$
$$f\ \mathbf{if}\ b\ \mathbf{then}\ x\ \mathbf{else}\ y\ \mathbf{fi} \ = \ \mathbf{if}\ b\ \mathbf{then}\ f\,x\ \mathbf{else}\ f\,y\ \mathbf{fi}$$

A function can be a bunch union,

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

and so can its operand. Function application is extended to non-element operands by base and distribution laws:

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

The range of function f is $f\,(\Box f)$.

Although the Function Application Law requires the operand to be an element, if a function uses its variable exactly once, and in a distributing context, then the function can be applied to a non-elementary operand because the result will be the same as would be obtained by distribution.

Operators on Functions

The operators $\wedge$ $\vee$ $+$ $\times$ have been defined for two 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 § 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, A and B are anything, and f and g are functions.

$$\wedge\langle x\colon null \rightarrow fx\rangle \ = \ \top \ = \ \wedge(A \rightarrow \top)$$
$$\wedge\langle x\colon e \rightarrow fx\rangle \ = \ fe$$
$$\wedge\langle x\colon A{,}B \rightarrow fx\rangle \ = \ \wedge\langle x\colon A \rightarrow fx\rangle \wedge \wedge\langle x\colon B \rightarrow fx\rangle$$
$$\wedge\langle x\colon \S f \rightarrow gx\rangle \ = \ \wedge\langle x\colon \Box f \rightarrow \mathbf{if}\ fx\ \mathbf{then}\ gx\ \mathbf{else}\ \top\ \mathbf{fi}\rangle$$

$$\vee\langle x\colon null \rightarrow fx\rangle \ = \ \bot \ = \ \vee(A \rightarrow \bot)$$
$$\vee\langle x\colon e \rightarrow fx\rangle \ = \ fe$$
$$\vee\langle x\colon A{,}B \rightarrow fx\rangle \ = \ \vee\langle x\colon A \rightarrow fx\rangle \vee \vee\langle x\colon B \rightarrow fx\rangle$$
$$\vee\langle x\colon \S f \rightarrow gx\rangle \ = \ \vee\langle x\colon \Box f \rightarrow \mathbf{if}\ fx\ \mathbf{then}\ gx\ \mathbf{else}\ \bot\ \mathbf{fi}\rangle$$

$$+\langle x\colon null \rightarrow fx\rangle \ = \ 0 \ = \ +(A \rightarrow 0)$$
$$+\langle x\colon e \rightarrow fx\rangle \ = \ fe$$
$$+\langle x\colon A{,}B \rightarrow fx\rangle + +\langle x\colon A\text{‘}B \rightarrow fx\rangle \ = \ +\langle x\colon A \rightarrow fx\rangle + +\langle x\colon B \rightarrow fx\rangle$$
$$+\langle x\colon \S f \rightarrow gx\rangle \ = \ +\langle x\colon \Box f \rightarrow \mathbf{if}\ fx\ \mathbf{then}\ gx\ \mathbf{else}\ 0\ \mathbf{fi}\rangle$$

$$\times\langle x\colon null \rightarrow fx\rangle \ = \ 1 \ = \ \times(A \rightarrow 1)$$
$$\times\langle x\colon e \rightarrow fx\rangle \ = \ fe$$
$$\times\langle x\colon A{,}B \rightarrow fx\rangle \times \times\langle x\colon A\text{‘}B \rightarrow fx\rangle \ = \ \times\langle x\colon A \rightarrow fx\rangle \times \times\langle x\colon B \rightarrow fx\rangle$$
$$\times\langle x\colon \S f \rightarrow gx\rangle \ = \ \times\langle x\colon \Box f \rightarrow \mathbf{if}\ fx\ \mathbf{then}\ gx\ \mathbf{else}\ 1\ \mathbf{fi}\rangle$$

$$\S\langle x\colon null \rightarrow fx\rangle \ = \ null \ = \ \S(A \rightarrow \bot)$$
$$\S\langle x\colon e \rightarrow fx\rangle \ = \ \mathbf{if}\ fe\ \mathbf{then}\ e\ \mathbf{else}\ null\ \mathbf{fi}$$
$$\S\langle x\colon A{,}B \rightarrow fx\rangle \ = \ \S\langle x\colon A \rightarrow fx\rangle\ ,\ \S\langle x\colon B \rightarrow fx\rangle$$
$$\S\langle x\colon \S f \rightarrow gx\rangle \ = \ \S\langle x\colon \Box f \rightarrow \mathbf{if}\ fx\ \mathbf{then}\ gx\ \mathbf{else}\ \bot\ \mathbf{fi}\rangle$$

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

$(\wedge(A \rightarrow e) = e) \;\ge\; (A \ne null)$
$(\vee(A \rightarrow e) = e) \;\ge\; (A \ne null)$
$\S(A \rightarrow e) \;=\; \mathbf{if}\ e\ \mathbf{then}\ A\ \mathbf{else}\ null\ \mathbf{fi}$

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

We distribute application of function f over solutions as follows:
$f(\S g) \;=\; \S\langle y{:}\, f(\square g) \rightarrow \vee\langle x{:}\, \square g \rightarrow fx{=}y \wedge gx\rangle\rangle$

Let f be a function with a non-null domain. Let g be any function. If gx varies directly with x, then
$\wedge\langle x{:}\, \square f \rightarrow g(fx)\rangle \;\ge\; g(\wedge\langle x{:}\, \square f \rightarrow fx\rangle) \;=\; g(\wedge f)$
$\vee\langle x{:}\, \square f \rightarrow g(fx)\rangle \;\le\; g(\vee\langle x{:}\, \square f \rightarrow fx\rangle) \;=\; g(\vee f)$
If gx varies inversely with x, then
$\wedge\langle x{:}\, \square f \rightarrow g(fx)\rangle \;\ge\; g(\vee\langle x{:}\, \square f \rightarrow fx\rangle) \;=\; g(\vee f)$
$\vee\langle x{:}\, \square f \rightarrow g(fx)\rangle \;\le\; g(\wedge\langle x{:}\, \square f \rightarrow fx\rangle) \;=\; g(\wedge f)$
And in most instances, $\ge$ and $\le$ can be replaced by $=$. Here are the distributive or factoring laws (omitting the non-null domain).
$\wedge\langle x \rightarrow -fx\rangle \;=\; -\vee\langle x \rightarrow fx\rangle \;=\; -\vee f$
$\wedge\langle x \rightarrow fx + y\rangle \;=\; \wedge\langle x \rightarrow fx\rangle + y \;=\; \wedge f + y$
$\wedge\langle x \rightarrow fx - y\rangle \;=\; \wedge\langle x \rightarrow fx\rangle - y \;=\; \wedge f - y$
$\wedge\langle x \rightarrow y - fx\rangle \;=\; y - \vee\langle x \rightarrow fx\rangle \;=\; y - \vee f$
$\wedge\langle x \rightarrow fx \wedge y\rangle \;=\; \wedge\langle x \rightarrow fx\rangle \wedge y \;=\; \wedge f \wedge y$
$\wedge\langle x \rightarrow fx \vee y\rangle \;=\; \wedge\langle x \rightarrow fx\rangle \vee y \;=\; \wedge f \vee y$
$\wedge\langle x \rightarrow fx \triangle y\rangle \;=\; \vee\langle x \rightarrow fx\rangle \triangle y \;=\; \vee f \triangle y$
$\wedge\langle x \rightarrow fx \triangledown y\rangle \;=\; \vee\langle x \rightarrow fx\rangle \triangledown y \;=\; \vee f \triangledown y$
$\wedge\langle x \rightarrow fx < y\rangle \;\ge\; (\vee\langle x \rightarrow fx\rangle < y) \;=\; (\vee f < y)$
$\wedge\langle x \rightarrow fx > y\rangle \;\ge\; (\wedge\langle x \rightarrow fx\rangle > y) \;=\; (\wedge f > y)$
$\wedge\langle x \rightarrow fx \le y\rangle \;=\; (\vee\langle x \rightarrow fx\rangle \le y) \;=\; (\vee f \le y)$
$\wedge\langle x \rightarrow fx \ge y\rangle \;=\; (\wedge\langle x \rightarrow fx\rangle \ge y) \;=\; (\wedge f \ge y)$
$\wedge\langle x \rightarrow \mathbf{if}\ y\ \mathbf{then}\ fx\ \mathbf{else}\ z\ \mathbf{fi}\rangle \;=\; \mathbf{if}\ y\ \mathbf{then}\ \wedge\langle x \rightarrow fx\rangle\ \mathbf{else}\ z\ \mathbf{fi} \;=\; \mathbf{if}\ y\ \mathbf{then}\ \wedge f\ \mathbf{else}\ z\ \mathbf{fi}$

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

$\vee\langle x \rightarrow fx \triangledown y\rangle \;=\; \wedge\langle x \rightarrow fx\rangle \triangledown y \;=\; \wedge f \triangledown y$
$\vee\langle x \rightarrow fx < y\rangle \;=\; (\wedge\langle x \rightarrow fx\rangle < y) \;=\; (\wedge f < y)$
$\vee\langle x \rightarrow fx > y\rangle \;=\; (\vee\langle x \rightarrow fx\rangle > y) \;=\; (\vee f > y)$
$\vee\langle x \rightarrow fx \le y\rangle \;\le\; (\wedge\langle x \rightarrow fx\rangle \le y) \;=\; (\wedge f \le y)$
$\vee\langle x \rightarrow fx \ge y\rangle \;\le\; (\vee\langle x \rightarrow fx\rangle \ge y) \;=\; (\vee f \ge y)$
$\vee\langle x \rightarrow$ **if** y **then** fx **else** z **fi**$\rangle$ $=$ **if** y **then** $\vee\langle x \rightarrow fx\rangle$ **else** z **fi** $=$ **if** y **then** $\vee f$ **else** z **fi**

Function Inclusion

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

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

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

$\langle n\text{: } nat \rightarrow n, n+1\rangle\ 3 \;=\; 3, 4$

Functions are sometimes classified as partial or total, and sometimes as deterministic or nondeterministic.

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

Here is a function that is both partial and nondeterministic.

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

The union and intersection of functions are defined by the following four laws.

$(f, g)\, x \;=\; fx, gx \qquad\qquad □(f, g) \;=\; □f\text{ ‘ }□g$
$(f\text{ ‘ }g)\, x \;=\; fx\text{ ‘ }gx \qquad\qquad □(f\text{ ‘ }g) \;=\; □f, □g$

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

$(f\text{: } g) \;=\; (□g\text{: } □f) \;\wedge\; \wedge\langle x\text{: } □g \rightarrow fx\text{: } gx\rangle$

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

$(f = g) \;=\; (□f = □g) \;\wedge\; \wedge\langle x\text{: } □f \rightarrow fx = gx\rangle$

Let suc be the successor function on the naturals.

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

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

$\quad (suc\text{: } nat \rightarrow nat)$ use Function Inclusion Law
$= \quad (nat\text{: } nat) \;\wedge\; \wedge\langle n\text{: } nat \rightarrow suc\ n\text{: } nat\rangle$
$= \quad \wedge\langle n\text{: } nat \rightarrow n+1\text{: } nat\rangle$
$= \quad \top$

And, more generally,

$(f\text{: } A \rightarrow B) \;=\; (A\text{: } □f) \;\wedge\; (fA\text{: } B)$

We can similarly show

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

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

$eat\text{: } nat \rightarrow (0, eat)$

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. Since its variable f is used exactly once in *check*, and in a distributing context (*even* distributes over bunch union), we can apply *check* to a functional operand (even though functions are not elements). 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 fewer results. 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 $\bot$.

Function Composition

Let f and g be functions such that $\neg(f\colon \Box g)$ (f is not in the domain of g). Then $g\,f$ is the composition of f and g, defined by the Composition Laws

$$\Box(g\,f) = \S\langle x\colon \Box f \to fx\colon \Box g\rangle$$
$$(g\,f)\,x = g\,(f\,x)$$

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 that $\neg(f\colon \Box h)$ (f is not in the domain of h), and $\neg(g\colon \Box(h(fx)))$ (g is not in the domain of $h(fx)$). Then

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

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 $\neg(f\colon D)$, 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

$$(+h)\ x \;=\; +(h\ x)$$
$$+\langle x \rightarrow hx\rangle \;=\; \langle x \rightarrow +(hx)\rangle$$

This enables us to write some distributive/factoring laws neatly; for example,

$$\wedge(-f) \;=\; -(\vee f)$$

Since the operator # does apply to a function, it cannot be composed with a function.

Two-operand 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,

$$(f \times g)\ x \;=\; f\ x \times g\ x$$
$$\langle x \rightarrow fx\rangle \times \langle x \rightarrow gx\rangle \;=\; \langle x \rightarrow fx{\times}gx\rangle$$

So the inner product of f and g is $+(f{\times}g)$. This enables us to write some distributive/factoring laws neatly; for example,

$$\wedge(f{\le}g) \;=\; (\vee f \le \wedge g)$$

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

(The following alternative to operator-function composition was considered and rejected. It would be more uniform to say that all operators compose with functions, so that

$$(f \times g)\ x \;=\; (f\ x \times g\ x)$$
$$(f = g)\ x \;=\; (f\ x = g\ x)$$
$$(f : g)\ x \;=\; (f\ x : g\ x)$$

and then, for functions f and g having the same domain

$+(f{\times}g)$	$= +\langle x \rightarrow fx{\times}gx\rangle$	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

But function equality and function inclusion are wanted so frequently that writing $\wedge(f{=}g)$ instead of $f{=}g$ is too great a price. And we would have no notation for functions that introduce function-valued variables.)

Selective Union

If f and g are functions, then

$f \mid g$ “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:

$$\square(f \mid g) \;=\; \square f, \square g$$
$$(f \mid g)\ x \;=\; \textbf{if}\ x{:}\ \square f\ \textbf{then}\ fx\ \textbf{else}\ gx\ \textbf{fi}$$

Selective union is idempotent, associative, and composition distributes over it.

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

Selective union gives us a way to express a function by listing domain-result pairs as in the following example:

$$0 \rightarrow 2 \mid 2 \rightarrow 1 \mid 1 \rightarrow 0$$

When the domain values are textual, we have the familiar “record” or “structure” from various programming languages; by letting the result values be bunches we have “record types”.

Lists as Functions

We have already presented lists; now we identify them with functions. If L is a list, then

$$L \;=\; \langle n\colon (0,..\#L) \rightarrow L\,n\rangle$$

The domain of list L is $0,..\#L$. The length of a list is the same as the size of its domain, and the same notation $\#L$ is used. 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. Lists can be composed with functions that are not lists. For example,

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

We can also mix lists and other functions in a selective union. With function $1\rightarrow21$ as left operand, and list $[10;\,11;\,12]$ as right operand, we get

$$1\rightarrow21 \mid [10;\,11;\,12] \;=\; [10;\,21;\,12]$$

just as we defined it for lists. And

$$[10;\,11;\,12] \;=\; 0\rightarrow10 \mid 1\rightarrow11 \mid 2\rightarrow12$$

Applying an operator to a list, $+L$ conveniently expresses the sum of the items of list L .

The Function Inclusion Law, specialized to lists, says that a list includes all its extensions.

$$[S;\,T]\colon [S]$$

For example, $[3;\,2;\,4]$ is not only a list of three items, but also all lists that start with those three items. The intuition is the same as for functions. If a context requires a list of three integers (perhaps because it will be indexed with 0 , 1 , and 2 , and these items will be checked for even or oddness), then any list beginning with three integers will do.

Another consequence of the Function Inclusion Law is that a string includes all its extensions.

Limits

We introduce three operators, $⩔$ (“liminf”), $⩓$ (“limsup”), and $\Diamond$ (“limit”) that apply to functions with domain nat . Here are their laws.

$⩔f \;=\; \vee\langle m \rightarrow \wedge\langle n \rightarrow f(m+n)\rangle\rangle$	Liminf Law
$⩓f \;=\; \wedge\langle m \rightarrow \vee\langle n \rightarrow f(m+n)\rangle\rangle$	Limsup Law
$⩔f \;\le\; \Diamond f \;\le\; ⩓f$	Limit Law

with all domains being nat . (We intentionally avoid all mention of “existence” of limits. If f happens to be a binary function, then $⩔f$ is traditionally called “eventually always f ”, and $⩓f$ is traditionally called “infinitely often f ”.)

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

$$\Diamond\langle n\colon nat \rightarrow n\rangle \;=\; \top$$

$$\Diamond\langle n\colon nat \rightarrow 1/(n+1)\rangle \;=\; 0$$

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

$$-1 \le\; \Diamond\langle n\colon nat \rightarrow (-1)^n\rangle \;\le\; 1$$

In general,

$$\wedge f \le ⩔f \le \Diamond f \le ⩓f \le \vee f$$

For nondecreasing f ,

$$\wedge f \le ⩔f = \Diamond f = ⩓f = \vee f$$

For nonincreasing f ,

$$\wedge f = ⩔f = \Diamond f = ⩓f \le \vee f$$

Here are some Distributivity or Factoring Laws for limits (traditionally called "continuity").

$$\Diamond(-f) \;=\; -\Diamond f$$
$$(\bot<\Diamond g<\top) \;\le\; (\Diamond(f+g) \;=\; \Diamond f + \Diamond g)$$
$$(\bot<\Diamond g<\top) \;\le\; (\Diamond(f-g) \;=\; \Diamond f - \Diamond g)$$
$$(0<\Diamond g<\top) \;\le\; (\Diamond(f\times g) \;=\; \Diamond f \times \Diamond g)$$
$$(0<\Diamond g<\top) \;\le\; (\Diamond(f/g) \;=\; \Diamond f \,/\, \Diamond g)$$
$$\Diamond(f\wedge g) \;=\; \Diamond f \wedge \Diamond g$$
$$\Diamond(f\vee g) \;=\; \Diamond f \vee \Diamond g$$
$$\Diamond(f\vartriangle g) \;=\; \Diamond f \vartriangle \Diamond g$$
$$\Diamond(f\triangledown g) \;=\; \Diamond f \triangledown \Diamond g$$
$$\Diamond(f\le g) \;=\; (\Diamond f \le \Diamond g)$$
$$\Diamond(f\ge g) \;=\; (\Diamond f \ge \Diamond g)$$
$$\Diamond(f<g) \;\ge\; (\Diamond f < \Diamond g)$$
$$\Diamond(f>g) \;\ge\; (\Diamond f > \Diamond g)$$

Now that we have defined $\Diamond$, we can define the real numbers as follows:

$$real \;=\; \Diamond(nat\rightarrow rat)$$

Notice that $nat\rightarrow rat$ includes all functions with domain at least nat and result at most rat , and we take the limits of all such functions. (This definition includes $\top$ and $\bot$ in $real$. Excluding them is possible, but uglier: $\S\langle x\colon \Diamond(nat\rightarrow rat) \rightarrow \bot < x < \top\rangle$.)

Conclusion

I have presented an algebra that unifies numbers with booleans, types with values, and 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 $\vee\langle x\colon D \rightarrow Bx\rangle$ is quite useful for saying "There exists an object, let's call it x , in the bunch of objects represented by D , such that B is true of x .". But when D represents an infinite bunch of objects, $\vee\langle x\colon D \rightarrow Bx\rangle$ differs slightly 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 more closely. Another way to resolve the discrepancy is to part from the traditional mathematical idea of existence in order to fit the algebra, or even abolish the idea of mathematical existence altogether. I prefer abolishment.

Acknowledgements

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

References and Sources

[0] E.C.R.Hehner: "from Boolean Algebra to Unified Algebra", *the Mathematical Intelligencer*, Springer, 26(2) 3-19, 2004, and www.cs.toronto.edu/~hehner/BAUA.pdf

[1] E.C.R.Hehner: *a Practical Theory of Programming*, Springer 1993, current edition available free at www.cs.toronto.edu/~hehner/aPToP