UNIT AND BOUNDARY ARITHMETICS
TENTATIVE TABLE OF CONTENTS

William Bricken
12/04/07

SECTION II

SECTION III

A Theory of Representation
- Form or Content?
  - Enactive Systems
  - Symbolic Systems
  - Iconic Systems
- Multiple Representations

How People Work
- Math is Messy

Information Processing
- Overview of the Model
- The Information Processing Model
- Models of Cognitive Processes
- Models of Error Processes
- Davis' Frame Model

Early Childhood Understanding
- As Easy As One, Two, Three
- Tension Between Abstract and Concrete
- Two Early Studies Define the Research Issues
- The Newman Hierarchy
- Gibb
- Ginsberg
- Elementary Algebra

TEACHING MATHEMATICS
- Something Is Not Working
- Spatial Math Is Manipulable
- Manipulatives
- Virtual Manipulatives
- Spatial Analogs

SECTION IV

SECTION V

UNIT ENSEMBLES
- Dot Pictures
- Foundations
- Void Foundations
- Counting vs One-to-one Correspondence
- Calculating without Counting
- Boundary Delineation
- Names and Variables

MEREOLOGICAL CONCEPTS
- Void
  - Void Meta-Notation
- Units
- Fusion
  - Variarity
  - Many-to-One
  - Flatness
  - Pre-associative, Pre-commutative
  - Fusion Relations
    - Axiomatic Structures
    - Self Structures
    - Structures with a Constant
    - Symmetry Structures
    - Ordering Structures
    - Universal Relation
    - Mixing Structures
- Discrete Spaces and Partitions
  - Partition Lattice

EQUALITY AND SUBSTITUTION
- Axioms of Equality
  - Functional Substitution
  - Logical Inference
- Pattern Substitution
  - Transforming an Environment
  - Types of Valid Substitution
    - Substitution of Equals
    - Global Replacement within an Equation
    - Functional Substitution Applied to Substitution
- Condensed Notation
- Special Cases
  - Self and Universe
  - Void Substitution
  - Value Maintenance

- Comparison of Axioms
- Logarithms
- Roots

STRUCTURE OF FRACTIONS

- Equivalent Fractions
- Multiplying Fractions
- Dividing Fractions
- Adding and Subtracting Fractions
  - Like Denominators
  - Unlike Denominators
- Decimal Notation

ALGEBRA

- Combining Like Terms
- Solving Linear Equations
- Solving Proportions
- Cross-Multiplication
- Quadratic Equation
- Pythagorean Theorem
- Systems of Equations

SECTION VI

BOUNDARY ARITHMETIC AND DEPTH-VALUE NOTATION
Place value as Syntactic Sugar
Unit Ensembles Expressed in Place-value Notation
Place-value Fusion
Place-value Substitution

Base-10 Notation
Canonical Form of Numbers
READING BOUNDARY INTEGERS
? INSIDE AND OUTSIDE
? SHARING SPACE
MERGING
BOUNDARY ARITHMETIC COMPUTATION

SECTION VII

Spatial Algebra
James Numbers
James Imaginary
...

SUMMARY AND TABLES

UNIT AND BOUNDARY ARITHMETICS -- SECTION IA October 1, 2008 10:55 PM
William Bricken
January 2007

THE UNARY CALCULATOR --and-- THE VISUAL BOUNDARY CALCULATOR (any base bmath)

===SKETCH DONE===

CONTENTS
PREFACE
ABSTRACT

Binary Computers (Digitality)
Entrenchment of Ideas
Loss of Humanity

PREFACE

Sections I,II, and III can be read independently.

Section I is an introduction to the spatial representations and conventions of unit ensembles. Spatial representations differ significantly from conventional token-strings in both concept and usage. It is the difference between words and pictures. Because spatial formal systems have been suspect within the mathematical community, we provide a rigorous formal model of representation and transformation of forms. The formal theory is of unit ensembles, which are collections of unitary marks, such as found within tally and stroke arithmetics. The axioms and theorems of unit ensembles are both simple and intuitive, resting primarily on the familiar algebra of equalities. We discuss fusion and substitution as the primary operations on two types of units, Solid and Hollow.

Section II begins an interpretation of unit ensemble arithmetic as conventional integers. We show how unit ensemble arithmetic differs from the conventional arithmetic of integers for operations and for inverses. We then present several theories that constitute conventional arithmetic, and their unit ensemble counterparts. Differences in the structure of comparison operations highlight the absence of conventional group theoretic structures within unit ensemble arithmetic. Differences in addition and subtraction operations emphasize conventional departures from the Additive Principle that sums are represented by their parts. The integration of multiplication into equational substitution shows that multiplication and division do not add conceptual overhead. We conclude by looking at theories of exponents, rational numbers (fractions), and solutions of simple algebraic equations.

Section III introduces depth-value notation, used in spatial systems in place of place-value notation. Unit ensembles are base-1; we provide base-2 and base-10 equivalents. Encoding the power of a base in depth of nesting creates the boundary integers. We convert unit ensembles to boundary integers, and show that the transformation properties and rules of unit ensembles apply without modification to boundary integers. Fusion in space continues to implement addition and subtraction, and substitution of units continues to implement multiplication and division.

ABSTRACT

INTRODUCTION

This monograph is an introduction to spatial arithmetic and algebra, both applications of the new field of boundary mathematics. Based on recent advances in reasoning and computation with spatial forms, these new methods of understanding mathematics unite intuition with abstraction. Spatial arithmetic helps to bridge the learning gap between concrete understanding and abstract symbol manipulation by rigorously simplifying the rules of algebra. Boundary mathematics provides concepts and tools that closely align the abstract formality of mathematical systems with our concrete experience of real objects.

The great majority of people in the Untied States do not understand mathematics. Math phobia is known to be rampant. More than half of all school children cannot pass the minimal proficiency exams in mathematics. As technology grows more prevalent, employers are demanding a more educated workforce, one that can understand and use the mathematical aspects of commerce and business. Blame has been passed around freely, to students, to teachers, to the mathematics curriculum, to State and Federal programs. Although there is failure at each of these levels, we propose that the common problem (what [ref] calls innumeracy) is the current structure of mathematics itself. In particular, since the early 1900s, mathematicians have embraced the idea that math, "formal" mathematics, must exclude intuition, psychology, and education. Mathematics, as a subject, has been defined to be independent of human skills.

Our suggested improvement addresses elementary mathematics, that content taught in Grades K-12. We are not suggesting a modification of advanced or engineering mathematics, those topics usually taught at the college level. There have been many suggested curriculum reforms for K-12 math, the changes suggested here are of a substantively different type. In almost all cases, suggested changes in the mathematics curriculum, in the way math is taught, and in the training and support of math teachers have accepted the established foundations of mathematics. Innovation is solely in terms of making understanding and teaching of the accepted foundations of math more efficient and more palatable. We too hope to improve understand and teaching, however we are suggesting that these improvements are possible by modifying the way we conceive of math itself.

No, one cannot change the nature of integers, numbers will always have the same abstract meaning. But the way we represent and manipulate numbers is a choice; there are many different representations and many different algorithms that achieve the fundamental operations of addition, subtraction, multiplication, and division. We are suggesting that both the formal foundations and the way we represent these foundations can be changed, without altering what numbers are. We are suggesting that the ways we formalize and manipulate integers should incorporate essential human needs such as intuition, experience, and interaction. The problem is not with our students or their schooling, it is with the presumption that mathematics must be unrelentingly abstract.

Platonism, Lakatos, Numez...

With the advent of modern algebra, the group theoretic properties of commutativity, associativity, zeros, and inverses have provided organization for teaching and learning both arithmetic and algebra. Group theory also defines and unites the diversity of possible algebraic structures into a single grand theory for modern algebra. The symbolic algebra approach trades the visual and physical intuition that arises from direct experience for the automated manipulation of structured strings of abstract tokens. Spatial algebra is a formal alternative to group theory for understanding and for teaching both arithmetic and algebra. It combines axiomatic rigor with simplicity of both representation and computation. However, in formalizing the concrete intuition of early childhood interaction with concrete objects, spatial algebra must step outside of the predominant conventional wisdom that defines modern algebra.

Such a divergence from the work of some of the most brilliant mathematicians on the planet cannot be taken lightly. However, mathematics is that particular subject that, due to its inherent abstraction, can and does accommodate significantly divergent ideas. The recent history of mathematical advance is punctuated with surprising, and at times traumatic, revolutions of concept and thought. The discovery of non-Euclidean geometries, the intractability of non-differentiable "monster curves" that were eventually recognized as fractals, the failure of formalism attributed to Godel's proof of the incompleteness of arithmetic, new approaches such as game theory, cellular automata, computational complexity, quantum logic and string theory, each of these drastic innovations of thought has enriched the recent growth of mathematics with deep conceptual reorganization. Perhaps only the exploration of the cosmos has exposed more assumptions and raised more questions.

Pragmatics

As mathematical understand has extended its scope, the abstraction and complexity of mathematics has moved further and further out of the reach of common understanding. And it appears that at least in the United States, even simple arithmetic and algebra have become more difficult for young students. Are our schools failing to keep pace with the rapid growth of technology, are our students becoming less intelligent, what is driving poor academic performance in mathematical subjects? We propose a possible reason for lower test scores, and a potential solution to math phobia. But, as should be expected, addressing such large and pervasive problems requires significant modification of the way we approach math education. We suggest that math has become less comprehensible because over the last hundred years, the agenda of professional mathematicians has explicitly sought to make mathematics less comprehensible. The concepts associated with symbolic formalism have been

pursued as a singular goal. These concepts include removal of intuition, of psychological necessity, of physical interaction, and of concrete understanding from the practice of mathematics. The formal agenda has contributed a tremendous capability for computers to do mathematics, at the loss of the capability of people to understand the mathematics being performed.

The field of mathematics has steadfastly refused to incorporate common accessibility to mathematical concepts, not by hiding information, but by constructing information without the guidance of psychology or education. Our schools are failing to teach math well because the math they are teaching is not constructed to be readily teachable.

Our objective is to reintroduce elements of human learning and human psychology into the structures of mathematics. In order to avoid the common attribution that such human elements are not math, we introduce a rigorous formal mathematical system for rational numbers that is also rigorously based on human capabilities. Math fails because it has forgotten, nay abandoned, its parents. It is now the concubine of computational machines. And most insidiously, rather than recognize that mathematics is a human activity, since the 1970s we have redefined the human mind to be a symbol processing machine.

Making Soup

Here is a simple example of constructing conditions for failure to learn mathematical skills. Imagine that you need to add two pints of water to a soup mix. In the best of kitchen worlds, you have a measuring cup that holds exactly two pints. In a slightly less perfect kitchen, you may have a one pint measuring cup. But you may unfortunately find that the only measuring cup is metric, holding say one liter. You can of course turn to the back of the cookbook, look up the conversion factors, compute how many liters you need, and measure accordingly. This mismatch of units contributes to the difficulty of making soup, but the problem can be overcome in two ways, one by doing some simple math conversions, and the other by relying on prior experience. If you have made soup before, you may have a memory of what two pints looks and feels like. You can call upon prior experience, and estimate how much you may need to fill the metric measuring cup.

We have traced this cooking example through several levels of difficulty. With the right tools, the soup will be successful. Even without the right tools, a little math or a little experience can come to the rescue. In the parallel world of a math classroom, a student may have to solve a long division problem, such as

$$349756 \div 432$$

This circumstance is faced by every sixth grade student. "Can I use a calculator?" Or does the student need to practice the intricate symbol manipulation designed specifically with the inner workings of a calculator in mind. Does the student need to learn to act exactly like a calculator, failing if some human slip may occur? Without the right tools, can a little math or a little experience come to the rescue?

The math classroom can be more insidious than the inconvenient kitchen, because in the classroom the right tools are available, their use though may be forbidden. At least kitchen utensils do not work against your desire to make some soup. How can a little math come to the rescue of the long division student? The skill of estimating an answer, of seeing almost immediately that the answer is around 800, is most usually discredited by math teachers as not addressing the problem. An exact answer is required, the type of answer that a hand-held calculator produces trivially. The essentially human skills of seeing through complexity and of knowing what is about right are denigrated. Instead, to get credit for doing the problem correctly, the student must act exactly like the calculator, not only using the wrong tool (a working mind) for the desired task, but also abandoning innate skills (a mind working) in favor of blind symbol manipulation.

And if a little math cannot come to the aid of our student, can a little experience be of help? Well, no. Symbol manipulation is not based on physical experience, it is based on memorization and automation. And here lies the main problem. Even accepting that the right tools are not available, even accepting that mathematical intuition is not the goal, our student must also perform manipulation skills that are not designed for human consumption. He must, figuratively, perform the long division with his hands tied behind his back. He must use a long division algorithm that does not align with his physiological skills, let alone his sensibility or intuition. Should we step back to a time before hand-held calculators, the issue still remains. The algorithms of long division, as well as those of each of the other basic operations of arithmetic, are both inconvenient and unconsciousable for use by human minds. That is why calculators were invented, that is why they are in use almost everywhere. Every grade school student know this.

One thing is well known about mathematics: there are always many ways of doing the same calculation. And that is because, at least in math, there is a profound difference between concept and implementation. Pure math is about abstraction and power. Once a route to a solution (or a proof) has been seen, the task then changes to mechanics, to following rules without error. Calculation is the realm of Computer Science. There, experts of computational technique devise data structures and algorithms and silicon circuits that excel at following the rules of symbolic computation.

Representation and Meaning

Mathematical knowledge is a densely connected matrix of interrelated concepts and representations. The dominant representation for printed mathematics consists of strings of arbitrary tokens [ref phildia]. The intended meaning of these token-strings is maintained by explicitly permitted structural relations within the one-dimensional arrays of tokens. Valid syntactic transformation of mathematical strings thus relies upon a formal theory of totally ordered token-strings. The semantics of these token-strings includes a rich and well-studied collection of basic mathematical structures, specifically predicate logic, set theory, integer arithmetic, and algebra.

Mathematical style strictly distinguishes between the complexity and subtlety of concept and the choice of representations of concepts. This is often referred to the semantics/syntax barrier. Syntax is form, semantics is meaning. We propose, however, that representation occasionally leads to structural commitments that do injustice to the abstract concepts being represented. Seminal mathematical concepts find their way into a conventional notation, and then the notation becomes entrenched. Our understanding of the concepts becomes clearer over time, but the representation is not revised to reflect this new-found clarity. We often end up with multiple forms of representation and multiple meanings for the same representation [ref kaput]. Needless to say, this makes teaching and learning mathematics quite difficult.

Simple Ideas Built With Complexity

"Multiplicity ought not be posited without necessity."
-- William of Occam (1340)

A mathematical approach looks toward simplicity, since mathematics is a common ground of communication, with well-understood structural relations expressed as well-defined strings of symbols. However, the foundation of mathematics established in the 20th century is itself complex, consisting of:

sentential logic -- the familiar AND, OR, NOT, EQUIVALENT, and IMPLIES

predicate logic -- a domain of sets referring to structured objects; constraints such as closure, validity and consistency; mappings over domains, and quantification of symbols.

set theory -- a collection of axioms that assure sets, loosely collections, of specific structures actually exist

integer arithmetic -- familiar numbers (that lead to quite unfamiliar types of numbers), includes recursion and iteration for processing, and induction for proving.

Recently discovered mathematical forms (such as fractals, substitution/reduction systems, and cellular automata) use innovative structural symbols and techniques to express mathematical form, each embodies a notion of complexity but not necessarily a notion of simplicity.

Pruning Complexity

It is unlikely that a newer and simpler way to reason and to compute will appear out of nothing. The approach taken here is to begin with simple integers, and to ask "Which aspects of the structural form are not really necessary to achieve the intended meaning of integers?"

This question is nearly equivalent to asking: "Which essential aspects of commonly accepted and validated mathematical forms have been adopted erroneously by the greatest minds of humanity?" These minds, and humanity itself, have been making choices that have lead us to the great edifice of mathematics that now informs the underpinnings of our digital/technological culture. Newer systems of mathematics continue to appear, expressing the evolution of mathematics itself. Some of these new ideas integrate seamlessly into prevalent mathematical structures, while others appear to be irreducibly different. [[ref smith's book? wolfram on computation everywhere]]. Herein we simply propose new ways to think about old math. And we propose new foundations based firmly on ancient mathematical practices.

In pursuit of evolutionary growth of mathematical structures, we propose to examine new classes of expressively-equivalent formal representation. The primary goal is to simplify structural axioms by recording them in a representation that requires less overall effort. During the course of developing computational efficiency, we will refer structural techniques back to common forms and practices, not only to provide a link of validity, but to suggest how to think about conceptual interpretations of the reduced notation.

Is there a formal basis for the elementary arithmetic of whole numbers that is more intuitive than that provided by Frege and Peano at the turn of the twentieth century? We explore the requirements for a modern axiomatization of the type of arithmetic taught to four-year olds in kindergarden. The perspective of seeking simpler foundations for common mathematical structures has both philosophical and psychological roots, however here we are guided by pragmatism. Generally, simplification of foundations does not erode the sophisticated elaboration that marks mathematical progress, but it may contribute insight into prospective new tools that can contribute to further refinement of thought.

Where is Simple Mathematics Hiding?

In later sections, we suggest techniques to incorporate simplicity into conventional foundational mathematics. Here are some quite basic technical constraints that can be evaluated for removal:

-- Tokens (words) must refer to unique, existent mathematical objects and mappings.
-- Every concept embodied within a formal system must have a representation.
-- Mathematical transformations are free of time and implementation cost.
-- Tokens can be freely replicated.
-- There are an infinite supply of variable letters.
-- Processes must be stepwise
-- Mappings must have a specific arity.
-- The formal manipulation of spatial structures is an informal technique to achieve the formal manipulation of strings.
-- A token for zero is required for efficient algorithms.
-- Place-value notation is the most efficient way to represent integers for computation.
-- One-to-one mappings are necessary for group structures.
-- Single steps must be taken to move from premise to conclusion.

Removal of some or many of these constraints is not intended as a reconstruction of existing mathematical foundations; it is intended to show a particular type of mathematical evolution, that of condensation of representation via removal of non-useful redundancy.

The suggested simplification techniques for mathematical representation and transformation that follow are related, in that they are derived from the use of abstract two-dimensional forms in place of strings. There is an underlying theme: string notations encourage sequential concepts, while spatial forms encourage parallel concepts.

The guiding light that will steer our attempts toward simplification of the foundations of integers is that of common understanding. There is no reason why the formal foundations of a universal tool such as mathematics should not be accessible to everyone who may need to use mathematics. And we believe that there is no reason that the subject matter to be understood, at least by K-12, should not be easy to understand.

What Comes Next

Ancient systems of arithmetic are believed to have emphasized counting rather than computation, and are generally considered to be algorithmically inefficient. Herein, we demonstrate that unit arithmetic can be highly efficient computationally, its inconvenience is solely in the act of counting, in determining the result of a computation. By liberating unit arithmetic from linear strings, that is, by expressing the forms and transformations of arithmetic on a plane, we provide a depth-value notation that achieves the same efficiency as place-value notation. The resulting extension to unit arithmetic is called boundary arithmetic. We also provide a textual format for depth-value notation that addresses the issues of compatibility with current string-based processing methods. The techniques of boundary mathematics [refs] have informed all structural choices presented herein.

===FIX
In Section 2, we outline the non-conventional approach of spatial, or boundary, mathematics that informs the syntax of unit and boundary arithmetics. Section 3 examines the structure of spatial addition. Section 4 similarly outlines the structure of conventional equational theory that enforces the semantics of unit and boundary arithmetics. The spatial algebra developed in these three Sections is independent of its interpretation for integers. However, this algebra is designed to map intuitively onto an interpretation as unit arithmetic. Numbers are ensembles of units that share the same space; addition is placing ensembles in different spaces into the same space; multiplication is substituting a replicated ensemble for each unit another ensemble.

The central sections of the monograph presents a formal theory of spatial unit arithmetic. We treat spatial structure as formal structure rather than as an informal notation that must then be brought into conformance with the representational conventions of token-strings. The construction of spatial forms is guided more by mereology than by set theory. The operations of arithmetic are closer to physical manipulation of patterns than to computational algorithms. And without the comfort of first-order logic and group theory, proof, as well, incorporates non-standard structure and technique.

In Section 4 we examine unit arithmetic and some domain constraints that it provides. Then in Section 5 we provide an axiomatic approach to unit arithmetic that does not employ set theory or modern algebra. Sections 6 and 7 call upon the axiomatics of Section 5, and the semantic and syntactic theory of Sections 2 and 3 to provide an axiomatic understanding of addition and multiplication in unit arithmetic. Section 8 generalizes the results of the prior Sections to subtraction and division in unit arithmetic.

Section 9 introduces a spatial, depth-value notation for unit arithmetic that provides a computational efficiency equivalent to that of token-based place-

value notation. Section 10 provides examples of boundary arithmetic computation. These Sections address the cognitive and computational pragmatics of boundary arithmetic. Finally, We then summarize the monograph in Section 11 and provide convenient tables that organize our results.
===

How Math Works

One of the most baffling questions is: why is it that mathematics is so unreasonable effective in describing and predicting reality? Answers to this range from "Underneath it all, the universe is mathematics!", to "Mathematics is a voluntary myopia, a filter we place over an incomprehensibly complex reality that makes it bearable!"

Regardless of mathematical philosophy, we do need to explain how a young Einstein could write down an equation derived from imagination that accurately predicts slight variations in the orbit of the planet Mercury. If mathematics were indeed a human invention, there is no reason to expect that the planet Mercury would behave accordingly. Fortunately, this question is one of Physics: Why do mathematical equations accurately characterize physical events?

The wonder that maths "works" is of the same nature as the wonder that our distinctly human languages can in fact describe what we commonly experience as events on the Earth. Math is a very small subset of language, that particular part where ambiguity is not permitted.

The question of how math works does not need to address the perennial philosophical questions about the natures of reality and experience. We are seeking only a surface description. How do we make math work? We are not seeking an ultimate understanding of the relation between mathematical abstraction and physical or metaphysical meaning. We are asking the structural question: what are the workings of mathematics?

==========

Essay: Write less than one page in response to this famous question:

Why is mathematics so unreasonably effective in describing and predicting reality?

Math concepts come from reality.
From years of discovery and quantification, well defined rules.
Objective and precise, can prove anything that is true, builds upon itself.
Simple, clear and correct. reasonable, applicable to real-life.
Build upon known principles, symbols and abstraction, what leads to wisdom?
The way to understand the Universe, sea of interrelated details, successful prediction, laws.
Makes a lot of assumptions and not precise, rules of the way reality works.
Humans avoid analysis and reasoning, reality comes through visionary senses.
An invention of human minds, doesn’t apply to feelings, may not apply outside our experience.
Determines patterns, simplicity of concepts, describes what we perceive.

Math truncates information, simulates world, based on probability.
Explains natural mechanisms, prediction helps survival, identifies structural stabilities.
Ineffective, misses non-linear relations and rapid changes and irregular shapes.
Encompasses objective information, misses intangibles and complexities, simplified view.
Removes emotion, rigid rule sets, scientific beauty (simplicity, harmony,
==========

Figure %%% shows several answers to the question about the "unreasonable" effectiveness of math. [[discuss]]

Well, this question too turns out to be very complex. Over the next few sections, we will examine a diversity in the types of mathematical structures, along with some current approaches to systematize this diversity. We will look at how math is recorded, the numerals that represent the abstract concept of number, and the characteristics of how we write math down. And we will look at how math came to be, its historical evolution, with particular emphasis on the fundamental changes in math over the last one-hundred years.

The thesis is that it is the ways that we have formalized the recording of mathematical ideas, and the self-imposed limits of what we can say using the language of math, is the source of the problem of folks not understanding math. People do not understand mathematics will not because they have failed in some sense, but because mathematics is inherently confusing, and this confusion can be removed without loss of anything essential mathematical. It is as if we have put on an extra heavy overcoat, and gloves and hats and earmuffs, in order to go out into the warm sunshine. The over-clothes have been defined as "appropriate attire", the only thing you can wear in order to be properly dressed. Yet as we swelter in discomfort, we are all saying below our breath "The Emperor has too many clothes."

Mathematics have evolved magnificently, yet just as we do not have to know the inner workings of a carburetor in order to start the car, we do not really have to know the inner workings of a mathematical process in order to understand the results that it generates. And this is generally true for all of advanced mathematics. Like a virtuoso playing the violin, we can appreciate the music, but we do not feel compelled to play virtuoso violin.

So we come to the question: what is the appropriate level of mathematics to know and to understand? What grounding should each citizen have in order to be well educated, and to perhaps learn more as required by work or by play? We have a somewhat radical answer: enough to use the appropriate tool to get a result that has already been physically estimated by intuitive means.

Formalization

A formal symbol system consists of a set of tokens and a set of consistent rules for transforming arrangements of tokens. The rules express ways in which complex expressions can be built out of elementary objects. They also specify permissible transformations between complex expressions. The transformations are performed on strings of symbols, without regard to what they mean. Formal transformation, then, can be achieved by pattern-matching without thinking about what is being computed. This may sound unusual, but it is the way we do all math. Say that you have five oranges and you add four more oranges. What we do is to write down (or think) 5 + 4. In adding five and four, we forget about the oranges, we just add the numbers, and then with the result nine in hand, we return to say that we have nine oranges.

There is a common aspect of formal systems, independent of their particular content. All formal symbol systems share a body of canons, or basic understandings, that define their formality. One version of these conventions is presented below. Students in mathematics classes are expected to conform to these canons. For the most part they do, even though the canons of formal systems are rarely mentioned in mathematics texts and are rarely made explicit in mathematics classrooms. Since these conventions are common to all formal systems, they form a basis which defines our understanding of how math works.

The Canons of Formal Symbol Systems

The canons of formal symbol systems are those conventions that guide our agreement about the semantics, or meaning, and the syntax, or representation, of formal systems. These canons define the meaning of formality. Any violation of one of these canons is an error. The canons provide a complete set of rules to motivate mathematical behavior.

The following list is loosely paraphrased from Spencer-Brown's Laws of Form, one of the few mathematics texts that addresses the assumptions underlying the use of formal symbol systems. There is another list of canons that apply to equations, and the idea of transformation. This list is presented in Figure %% %, but not discussed until section %%%.

==========

Social Commitment and Intention

- New rules or conventions must be mutually agreed upon.
- What is not explicitly permitted is forbidden.
- Formal constraints are by choice.

Social Communication

- The representation of a concept is not inherently confusing.
- Symbols represent concepts that can be understood.
- Contraction of reference:

Representations may be abbreviated to any extent that is understood.

- Expansion of reference:

New representations may be introduced without limit, provided that confusion is avoided.

- A demonstration rests in a finite number of steps.

Relevance

- Universal properties within a system need not be indicated.
- A mutually agreed upon interpretative context is assumed.
- If a property is common to every indication, it need not be indicated.
- Meaning is indicated overtly in the form of expressions.
- The form of an expression indicates the choice of a meaning.

Representation

- Form: An expression is represented by its form.
- Content: The value of an expression is the value of its form.
- Value: Expressions with the same value can be identified.
- The interpretation of a symbol does not change independently of the interpreter.
- Reinterpretation denies the original interpretation.

==========

[[discuss]]

We first examine the inner workings of math, the 21st century idea of what it means to be mathematical.

Symbolic Abstraction

Symbolic mathematics uses symbols to express mathematical concepts. Symbols are abstract representations for some actual object, although in mathematics that object could be real, like an orange, or itself abstract, like a number. In the English language, a symbol can have a wide variety of referents. A flag can be a symbol, an action can be symbolic, and a parade can symbolize support for a cause. In mathematics, symbols are almost always tokens. A token is a letter or a number or a character, some form of typographical entity. Token symbols can be chosen arbitrarily, their structure bears no resemblance to their meaning. In fact, the intent of token symbols is to provide structures that can be manipulated by pattern-matching without concern for meaning.

A primary theme of this monograph is that mathematical symbols other than token-string can serve as computational objects. That is, it is certainly possible to

express formal pattern-matching and substitution rules using representations other than tokens. This theme, at least to us, appears to be very reasonable.

==========

Fig: patterns of rectangles and a rigorous substitution rule

```
            [][] = []

            [[]] = ><

      reduce   (spatial)
                  [ [[][]] [[]] ]
                  [ [[]  ] [[]] ]
                  [   ><   [[]] ]
                  [   ><    ><  ]
                  [             ]
```

==========

Fig %%% provides an example. A pure mathematics consists of formal structural rules, and no interpretation, or meaning. Certainly the pure mathematics of rectangles sharing space and rectangles nested within one another is as well-defined as any mathematics of token-strings. And if we were sufficiently pragmatic to also be interested in an interpretation, there is one for this spatial system, since it maps onto propositional logic [ref], itself the foundation of all formal math. FIg %%%

==========

Fig: Interpreting patterns of rectangles as propositional logic

```
            [] = True

            >< = False
```

Tables for Sharing Space and Nesting

```
            [][] = []

            [[]] = ><
```

==========

We have provided a simple example of a rigorous spatial system that also shows that core structures in conventional mathematics can be expressed using a spatial notation. Why then do mathematicians soundly dismiss spatial forms?
===fix

For example, a drawing of a triangle should be permissible as a symbol. And indeed it is, if we treat the drawing as a single token, with no internal structure. But this is not the intention of having a triangle serve as a symbol. We want the triangle to be an icon, a form that resembles what it refers to. Iconic mathematics uses icons, graphs and diagrams to express mathematical concepts. Thus, iconic math is a visual presentation of the meaning of math.
===

Mathematical Symbols and Icons

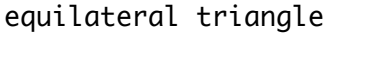
===
smooth
===

Pure mathematics can be considered to be the study of token structures, with no attempt to relate these structures to any usage, interpretation, or application. Symbolic mathematics is the formal study of token-strings. Iconic mathematics is the formal study of iconic forms.

A symbol is an association between an actual object or concept and an arbitrary token. We take advantage of this arbitrary choice by using symbols composed of letters and punctuation-marks, tokens that fit nicely into lines, thus providing display convenience.

An icon is an association between an actual object or concept and a token that resembles that object or concept. Thus, part of the meaning of an icon is its specific shape.

Icons are preferable to symbols since icons convey more information. The Arabic numeral 3, for instance, lacks the visual cardinality of the Roman numeral III. Some mathematical forms are inherently iconic, such as an equilateral triangle.



A formal icon can be translated into tokens, such as the words "equilateral triangle", or symbols derived from labeling the parts of the triangle: a = b = c, <AB = <AC = <BC. The icon, however, carries more information, since it illustrates as well as describes.

Symbolic description is easily standardized. More importantly, we have developed calculi to manipulate symbols (look-up tables, substitution of equals, logical deduction, term rewriting, theorem proving), permitting automated computation. Calculating with icons is thought to be difficult and clumsy.

However, several iconic calculi have recently been studied extensively; these include fractals, cellular automata, particle diffusion, and knot theory.

The triangle icon above refers to a mathematical object that cannot exist in physical space except as an approximation. We readily accept representational forms, such as photographs, of physical objects that do exist. A drawing can, in contrast, resemble an object that does not physically exist. In the case of mathematical objects, we are willing to believe in the existence of non-physical objects. The ideal equilateral triangle is assumed to exist, it is easily visualized. But it is not easily calculated with.

The problem is that a triangle figure is always of a specific triangle. The iconic triangle is general, referring to all triangles. When we say that the sum of the interior angles of a triangle is equal to 180 degrees, we are referring to all triangles, whether they are equilateral or isosolese or acute or obtuse. But the drawing of the triangle is only one of these. Thus, triangle drawings are not considered to be generic mathematical objects. The risk being guarded against is that we may build intuition based on one particular triangle, that turns out to be wrong when applied to another triangle.

We could however let a triangle stand in place of the concept Three, the three sides always representing the idea of threeness. This notation may be clumsy, but it is an example for which properties of the icon visually represent the abstract number concept. And the representation is not limited to a specific number, since an n-sided figure could be an iconic representation of n.

Symbolic Representation

Voice and text both unfold in a sequential stream over time. This structure requires symbolic codes that do not mimic their intention. Symbolic codes enforce a division between representation and reality, between mind and body, due to the arbitrary nature of their forms. Thus, words and actions apply to quite different domains.

Words are linear and symbolic. Symbolic representation has been accepted by the mathematical community at least since Descartes. It is currently an absolute standard for mathematical communication. Rigor is better served through a unified encodement; the symbiotic union of logical proof and mathematical notation, homogenized through digital computation, is a basis that assures manageable formal processes.

The language of concepts is not necessarily linear or symbolic, as we know from art, theater, film, music and architecture. Expression in these fields is

iconic; intention is conveyed through vision and experience rather than through trained memory and following rules.

Symbolic representation, like any encodement, imposes characteristic limitations on the form of expression, in exchange for a commonly learnable code. Teaching the linear symbolic code (reading, writing and arithmetic) is the primary activity throughout educational institutions.

The limiting presumptions built into symbolic mathematics require explicit permissions to overcome. These limits include:

- arity (counting):
  The number of objects that an operation can address is specific and fixed.

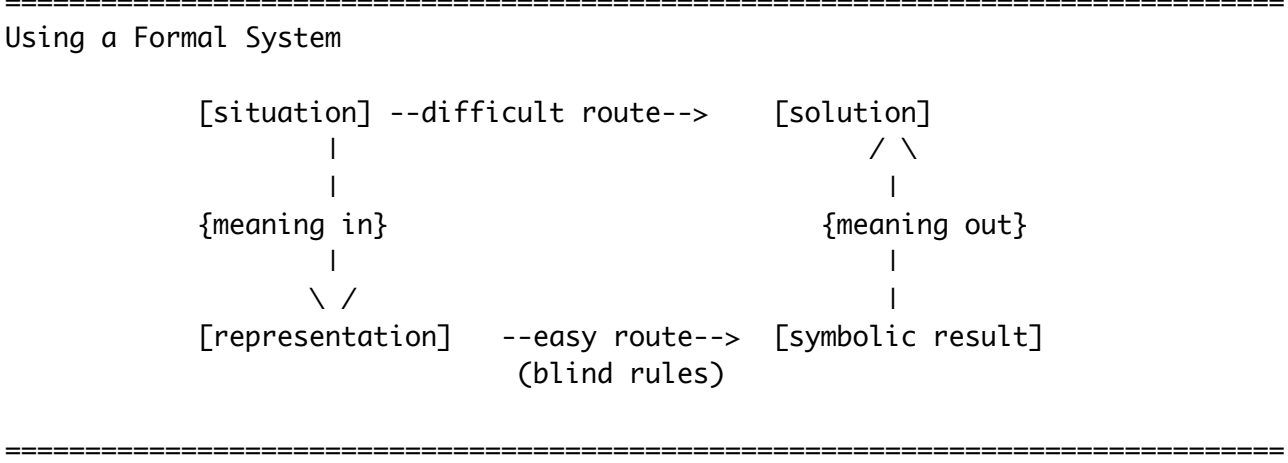
- associativity (grouping):
  The grouping of multiple objects imposed by specific arities is important.

- commutativity (ordering):
  The order of applying an operation to multiple objects is important.

Iconic formal systems express concepts in two and three dimensions, and can be shown to be equivalent to linear, one dimensional representations. Iconic representations do not naturally impose structural rules for grouping, ordering, and counting.

Using Math

Figure %%% provides an overview of how mathematics is made to work.



We begin with a situation that requires a solution. We could directly manipulate the situation itself until it resembles a solution. This usually happens with interpersonal and political situations. No every situation is amenable to the mathematical approach. In fact, no situation that requires physical action can be replaced by mathematics. What math can provide is guidance, it can help to determine which physical actions are likely to arrive at the desired solution.

The first step in using math is to recognize that the situation can be transcribed into abstract symbols, that we can map between meaning and representation. This is called mathematical modeling, and the process is often called encoding.

"We now come to the decisive step of mathematical abstraction: we forget about what the symbols stand for. ... There are many operations which [the mathematician] may carry out with these symbols, without ever having to look at the things they stand for."
-- H. Weyl

Assuming that the formal system accurately represents the actual circumstance, the next step is to blindly manipulate the symbols, rearranging them into a configuration that is simpler. This is called computation. When the configuration of symbols is sufficiently simple, we stop the computation, open our eyes, and use the mathematical model in reverse to get back to the actual situation. This is called decoding.

[quote]

It is indeed wonderous that this should work at all, let alone that it works for the most complicated physical processes we can imagine. All of Physics is mathematics.

[quote]

Math Is Art

===
Beauty

Scientific beauty consists of
1. simplicity (completeness, economy)
2. harmony (symmetry)
3. brilliance (clarity, connectedness)

"Beauty is the primary standard for scientific truth."
-- Augos and Staneiv, The New Story of Science, p.39

"You can recognize truth by its beauty and simplicity"
-- Richard Feynman

"Frequently a theorist will throw out a lot of data on the grounds that if they don't fit an elegant scheme, they are wrong."
-- Murray Gell-Mann

"A theory is more impressive the greater the simplicity of its premises is, the more different kinds of things it relates, and the more extended is its area of applicability."
-- Albert Einstein

===

There are, however, some significant caveats, in particular, determining whether or not the mathematical model accurately models reality. And these caveats are two directional. Does the math represent what is real, and do we define what is real to fit the math?

[quotes]

We will examine this process in detail, in order to find out just what is going on. Although a dangerous question, we first must ask: How does the human mind work? The answer is quite easy, we don't really know. One theme that we will discuss a few sections from now is that the mind definitely does not work like a computer. This issue is important because since the early 1970s, the vast majority of academic disciplines have actively explored this issue, valiantly seeking to establish the Information Processing Model of cognition, that the brain and a computer are just different devices (soft and hard systems) that do the same thing, process information. Accompanying this effort has been an extreme adoption of computer language to describe biological processes. We believe that this effort is one of the most significant acts of dehumanization in history. And in particular relation to the theme of the monograph, the excuse that permits teaching mathematics that is incomprehensible. To understand why, we will later look deeply at what it means to "process information".

The quick summary is mentioned above, information processing, mathematical abstraction, is blind to thought. The school of mathematical thought that is dominant today is called formalism. Formalism defines mathematics as a set of structural transformations that are not interpreted. About the same time that formalism began to gain sway in mathematics, around the turn of the twentieth century, behaviorism in psychology became dominant. Behaviorism is the idea that we should observe a human's physical actions, and completely ignore the human mind. Although behaviorism gave way over the century to a more compassionate view of psychology, formalism in mathematics just became stronger. The mathematics is free to ignore, and even to deny, the workings of the human mind is the specific cause of our countries failure to understand math.

The Modeling Hierarchy

The first step, that of encoding reality into a mathematical model, is actually several steps of abstraction. Fig %%%.

Conceptualization	(imaginary, perceptual, cognitive, real world)
Mathematical Model	(formal, symbolic, abstract, mathematical)
Data Structure and Algorithms	(representation, computational, software)
Machine Implementation	(actual, structured, physical hardware)

Fig: the Modeling Hierarchy

[[discuss]]

There are many forms of Conceptualization: art, music, film, architecture and sculpture, games, and mathematics. Here we will examine formal conceptualization, reducing an idea to a mathematical system.

Formal Modeling

A formal system (a mathematical system) consists of

1. several sets of labels (for objects, functions, relations)
2. rules for building compound sentences (or equations or expressions)
3. rules for evaluating and simplifying compound expressions
4. some axioms or assumptions which assert equivalence sets.

Formal = Atoms + Forms + Transforms + Axioms

Within this scheme, the innovation of spatial mathematics plays a small part. We are simply suggesting that the labels can be icons as well as tokens. Not a large change, but significant since iconic labels permit the following steps of building equations and rules and assumptions to be in a language that is closer to experience.

Here is a description of formal symbol systems that parallels the one above, but in a slightly more mathematical language.

A mathematical system consists of

1. a universe of discrete, stable, unique, disjoint, localized objects

2. a collection of stable states, or object configurations

3. stable maps between objects; a territory composed of objects and connections between objects (relations)

4. an interpretation that maps symbols/tokens one-to-one onto objects/meanings, together with a defined and stable notion of truth.

[[discuss two definitions, how components align, introduce State Space, a graph of state and transitions]]

From Conceptualization to Formalization

Here's what we do when we build a formal model (or do a computation):

0. Identify a collection of objects/events in the real world. This is the semantic mapping, how math is linked with reality. The objects/events must have these properties:

unique	not confused with different objects/events
stable and permanent	not in flux or changing too rapidly to identify
discrete	not lacking well defined borders
comprehendible	not confusing or too ambiguous
relevant	not outside of what we consider to be the objects in question
permitted,	not in violation of tacit understandings about how things are

1. Use unique labels to identify each of the things in the semantic mapping. The value of a label is the thing it identifies.

2. Limit our interest in the types of things in the real world to an abstract mathematical property, such as Truth or Count or Membership.

3. Use different labels to name different abstract things:

labels for things	object labels
labels for an entire set of things	property labels
labels for an arbitrary thing in the set	variable labels
labels to name properties of things indirectly	function labels
labels to name combinations of things	relation labels

4. Follow the rules of symbol transformation in manipulating the labels as if they were the things. However, the labels do not have to share any of the real world properties of the physical things.

In summary, we convert from physical to virtual, ignore the physical aspects of reality, manipulate the virtual (or digital) aspects using the rules of virtuality, and then return to physical reality with new knowledge.

Numeric or Symbolic Computation

===
clean up
numbers used for "measureables"
no longer about numbers
===

Compute symbolically, unless no efficient symbolic technique is known;
      then use optimized numeric techniques.

```
SYMBOLIC:
      meaning
            -- written as -->
                  symbol structures
                        -- reduced by -->
                              symbolic transformation rules
                                    -- turning into -->
                                          simpler symbol structures
                                                -- read for  -->
                                                      meaning

NUMERIC:
      meaning
            -- exemplified by -->
                  selected instances
                        -- substituted into -->
                              symbol structures
                                    -- reduced by -->
                                          numeric simplification rules
                                                -- turning into -->
                                                      approximate results
                                                            -- read for -->
                                                                  meaning
```

Mathematica

We have reached the culmination of the formalist agenda in the work of Stephen Wolfram.

[[discuss wolfram, mma, etc]]

Mma computes symbolically unless either
1. no efficient symbolic technique is known, or
2. processing efficiency is far more important than coding efficiency.

Otherwise, it uses optimized numeric techniques.

The Mathematica Program

A general purpose computational engine for
numerical calculations (arithmetic)
symbolic transformations (algebra)
graphic display (geometry)

A modern programming language with multiple styles
procedural
functional
logical
object-oriented
rule-based
mathematical

The Philosophy

The programmer's time is more valuable than the processor's time.

Thus, the architecture is
interpreted (interactive)
real-time
goal-oriented

"Programs you write in Mathematica may nevertheless end up being faster than those you write in compiled languages" p.506

- Processing speed depends on the exact implementation algorithm
- Mathematica algorithms are both sophisticated and optimized
- The internal data form is optimized and compiled for efficiency

Everything is an Expression

```
x + y              Plus[x,y]
120                Integer[120]
2ab                Times[2,a,b]
{a,b,c}            List[a,b,c]
i = 3              Set[i,3]
```

```
x^2+2x+1          Plus[ Power[x,2], Times[2,x], 1]
```

An undefined symbol is itself, providing functional transparency and WYSIWYG debugging.

The Meaning of Expressions

`F[x,y]`	`F` is the head. `x,y` are the contents. Apply function `F` to arguments `x` and `y`. Do action `F` to objects `x` and `y`. The label `F` points to elements `x` and `y`. The object-type `F` has parts `x` and `y`.

The head can both act on its contents (as a function) and maintain the structure of its contents (as an object), depending on context (the location of the expression, the presence of a definition).

Unity of Programming Paradigms

Mathematica accepts code in all of the modern programming paradigms.

“All the approaches are in a sense ultimately equivalent, but one of them may be vastly more efficient for a particular problem, or may simply fit better with your way of thinking about the problem.” p.487

“As a matter of principle, it is not difficult to prove that any Mathematica program can in fact be implemented using transformation rules alone.” p.503
===

The Virtues of Mathematical Models

[ref Gries and Schneider, A Logical Approach to Discrete Math]

- A mathematical model may be more understandable, concise, precise, or rigorous than an informal description in natural language.

- Answers to questions about an object or phenomenon can often be computed directly using a mathematical model of the object or phenomenon.

- Mathematics provides methods for reasoning:
  for manipulating expressions,
  for proving properties, and
  for obtaining new results from known facts.

This reasoning can be done without knowing or caring what the symbols being manipulated mean.

[[discuss]]

The Trouble with Mathematical Models

[[elaborate]]

- Only a small portion of the world and of our experience can be discretely objectified.

- Abstraction discards information.

- Modeling does not reflect human processes.
  (Students taught how to think in models generally make poor programmers.)

- Modeling dictates a worldview which, at times, may be dysfunctional.

"Present mathematical and scientific education is a hotbed of authoritarianism and is the worst enemy of independent and critical thought." -- Lakatos

- Human reasoning is physiologically mediated by human emotion. Discovery is intuitive and involves guesswork.

Math Is the Battleground human/tech

Modeling Constraints

Abstraction Constraints

===
improve transition
===

Growth of Algebra

==========
--CANONS TO GO WITH EQUALITY--
INTENTION
- Permissible transformations are specified by rules.

SUBSTITUTION
- In any expression, any arrangement can be exchanged for an equivalent arrangement.
- Equivalent intentions are expressed by equivalent symbols.

SIMPLIFICATION/TRANSFORMATION
- Following the rules does not change the meanings.
- Meaning is preserved when steps are taken.
- Symbols may be manipulated within the rules without altering their assigned interpretations.

EQUIVALENCE OF TRANSFORMATION
- The product of transformation steps is independent of the choice or sequence of steps, given a consistent result.
- All transformation paths leading to the same result are equivalent.

THEOREMS OF REPRESENTATION
- Agreement: Simplification is unique.
- Distinction: Rules specify permissible transformation steps.
- Steps do not change the meaning of an expression, although they do change the form.
- If the value of two expressions differ, then steps taken on the two expressions maintain that difference.

PROCEDURAL THEOREMS
- Identity: Expressions with identical forms represent identical values.
- Consequence: Expressions equivalent in value to a given expression are equivalent to each other.

CONNECTIVE THEOREMS
- Invariance: If a distinction distinguishes equivalent expressions, it is not a distinction.
- Variance: If different expressions each contain the same arrangement, then this arrangement can be communicated to the interpretative context of the expressions.

The use of algebraic equations and the EQUALS sign in the rules of boundary logic carries with it a substantive set of built-in assumptions. Algebra in general provides the familiar transformation techniques of replacement and substitution. Equations define constraints on variables in expressions. Algebraic representation places requirements on the use of variables; for instance, naming must be unique and names can stand in place of arbitrary expressions.

The EQUALS sign identifies equivalence classes of expressions. It has the properties of identity, transitivity, and commutativity which support the validity of the algebraic transformation techniques. Importantly, EQUALS is a Boolean connective, the value of an assertion of equality is either True or False.
===

Prior to 1900

computers commerce as algebra

Hilbert's Program

Mathematics and philosophy have always been inextricably entwined. Two and a half millennia ago, the Greeks introduced the revolutionary idea that fact should be based in proof, and that proof grew from the application of mathematical consistency. This attitude did not replace the dominance of the gods, but it did ascribe to them a preferred subject matter. Thus Descartes asserted that rationality was placed within us by God, and manifest by the a priori reasonability of numbers and logic to our minds [ref]. As the seventeenth century drew to a close, the work of Newton crowned the two thousand year drift of the source of truth from the Greek's observation and intuition to the Cartesian abstract concept supported by symbolic rather than physical structure. [redo]

Of course, God was never absent, but his Interests changed from the maintenance of every event in the experience of mankind into the maintenance of only mathematically structured events on the boundaries of the experiences of mankind. Science, rather than theology, was to step up to define God's Intent for His Glorious Machine. And since God was both Wise and Efficient, mathematics became the science of stating What Was in the simplest, most efficient language. The structure of the Universe slowly convolved into the structure of symbolic forms, that, presumably by a mapping provided by God, allowed us to know the Truth of creation.

As Mankind grew more fascinated by the mind of man, Kant reified reason itself. Truth, once divine, found itself entombed in thought, thought, that is, that was deemed reasonable by those who thought well.

==========

Kline p179 "Instead of starting with the whole numbers and fractions, and then taking up the irrational numbers, the complex numbers, algebra, and the calculus, the mathematicians tackled these subjects in the reverse order. they acted as though they were reluctant to tackle what could well be left alone as clearly understood and only when the need to logicize a subject was imperative did they undertake to do so. At any rate, only six thousand years after the Egyptians and Babylonians began to work with whole numbers, fractions, and irrational numbers, the mathematicians could finally prove that 2+2=4."

==========

Virtuality

===
Virtuality has become so prevalent in the current Information Age that much of what we do is never part of physical reality in the first place. Computer Science, for example, is a discipline in which the only connection to physical reality is silicon hardware. Study of the physicality of hardware is a different discipline, Electrical Engineering.
===

Binary Computers (Digitality)

Entrenchment of Ideas

Loss of Humanity

UNIT AND BOUNDARY ARITHMETICS -- SECTION IB October 1, 2008 10:55 PM
William Bricken
January 2007

Math Sensibility
- How Integers Work
- Measurement
- The Limits of Arithmetic
  - BASIC ARITHMETIC SENSIBILITY TEST
- Addition via Change of Perspective

What Is a Number?
- Numerals vs Numbers
- Reading and Computational Ease
- Computer Implementation

Our Conventional Number Systems
- Types of Numbers
  - Indicative Spaces
  - Nominal (Categories)
  - Ordinal Numbers
  - Interval Numbers
  - Rational Numbers
  - Cardinal Numbers
  - Truth Values
  - Unit Ensembles
  - Cyclic Number
  - Probabilities
  - Real Numbers
  - Imaginary Numbers

HUMANKIND'S GREAT ACHIEVEMENT (base system, positional notation)
- Hindu-Arabic Place-notation
- Uniform Base Systems
  - Base-1
  - Base-2
  - Base-10
  - Decomposition Strategies
- Explicit Zero

Math Sensibility

We have examined how math works. First we make a match between something real and something that represents what is real. If the structure of the real something conforms to the structures built into a mathematics, then we can use that mathematics to model the something real. Once the encoding step is completed, we are thoroughly within the world of math. We forget about what is real, and manipulate the mathematical structures, blindly following exact prescribed rules of transformation. Some of those transformations may be trivial, such as adding 1 and 1, and some may be extremely complex, such as predicting the weather. We saw that prior to about a century ago, mathematics was most about numbers, and that recently it has become about arbitrary symbolic structures. Once the computation is completed, whether using numbers or abstract symbols, we exit the world of math and return to reality by decoding the result of the computation. This result tells us something about real problem, hopefully something simpler, like the sum of 2 or whether or not we expect it to rain tomorrow.

We saw that the constraints on the real world situation that make it susceptible for mathematical analysis include both modeling constraints (choosing the right characteristics of the real world situation that match the structures of math) and abstraction constraints (choosing not to see some details of the real world).

Now we ask: Just how much of reality fits into the mathematical model? Actually, we will ask a much simpler question: Just how much of reality fits into the simple acts of addition and multiplication that are taught in lower elementary school? Should we be teaching children that simple arithmetic is very handy for lots of things, or should we be buffering that teaching with the ways that simple arithmetic is misleading?

How Integers Work

Measurement

The Limits of Arithmetic

We now consider how arithmetic may get us into trouble. In the spirit of math classes, we offer a Basic Arithmetic Sensibility Test. A dozen questions that appear to be easy applications of basic arithmetic, but that lead to false knowledge when arithmetic is applied. [refs]

==========

BASIC ARITHMETIC SENSIBILITY TEST

1. When is each of the following statements TRUE?

1a. 3 + 4 = 7
1b. 9 + 5 = 2
1c. 1 + 1 = 10
1d. 1 + 1 = 1

11. John put one lump of mud together with another lump of mud. How many lumps of mud does he have?

7. One cup of sugar plus one cup of water yields how many cups of sugar water?

2. My car gets 20 miles per gallon when traveling at 50 mph. How many miles per gallon will it get when traveling at 100 mph?

3. Leonardo DaVinci painted the Mona Lisa in 1550. If one Mona Lisa painting is worth $40,000,000, how much are two Mona Lisa paintings worth?

4. One barrel of oil costs $50. How much does one trillion barrels of oil cost?

5. A math teacher gives you one star for passing the final exam, one star for passing the midterm, and one star for reading the textbook. If three stars gets an A, what does two stars get?

6. The local supermarket sells one can of tuna for $.60 plus a special sale coupon. How much does two cans of tuna cost?

8. Sally's bank does not permit overdrawn accounts. If she has $100 in her account, and she writes a check for $200, how much will be left in her account?

9. A short intelligence test has three questions. It gives you one point if you correctly answer a question a George Washington, one point for a correct answer about polar bears, and one point for a correct answer about the area of a circle. What is your intelligence if you get all three questions wrong?

10. One person can paint a room in one day. Another person can paint the same room in three days. How long will it take to paint the room if they work together?

12. A professor lost half of his hair by the time he was thirty. When will he lose all of his hair?

==========

[[discuss each]]

What Is a Number?

Addition via Change of Perspective

How we think about mathematical concepts is often constrained by our representation of those concepts. Syntax and semantics (representation and concept) are tightly connected. The addition operation, for example, is conceptualized as binary when written in linear text:

```
x + y
```

To add three numbers, we must use two addition operators:

```
x + y + z
```

Column addition, however, reconceptualizes the addition operation to be variary (one operator can be applied to an arbitrary number of arguments):

```
  w
  x
  y
+ z
----
```

Naturally, the addition algorithms and techniques taught to students differ for the different representations.

The traditional representation of binary addition is one-dimensional. There are two locations for arguments, one on either side of the textual operator. Column addition increases the dimension of representation to the plane; digits of individual numbers are expressed horizontally, different numbers are expressed vertically. From a spatial perspective, the number of arguments that can be added in one operation depends upon the dimension of the representation.

symbolic
push together
shift perspective

===

Benacerraf --

{}, {}{}, {}{}{}, ... Not sets, but can be encapsulated

{}, {{}}, {{{}}}, ... Equal if in syntactic correspondence to itself

{}, {{}}, {{},{{}}}, ... Equal if elements in one-to-one correspondence

Three varieties of "numbers as sets", all are equally well grounded. So either:
1) all are correct, and the set-based representation of numbers is not unique; that is, sets are not fundamental to numbers,
2) one is correct and the others can be shown to be wrong,
3) none are correct; that is, numbers are not based on sets.

"There is no unique set of objects that are the numbers." p291

Numbers are essentially names that are in a sequence. It is not the object that forms "numbers", but the particular relation between objects that establishes Total Ordering
-- reflexive
-- antisymmetric
-- transitive

"Any object can play the role of 3; that is, any object can be the third element in some progression." p291 "The number words do not have single referents."
===

Numerals vs Numbers

"I know how to establish a one-to-one correspondence between numbers and objects, and I know how numbers work together to add and multiply, but I cannot for the life of me figure out how we discovered their names."

Reading and Computational Ease

Different representations of number systems have different characteristics (see chart below). The primary tradeoff is between readability and computability. In prehistory, unit arithmetic provided easy computing, but reading a result required counting the resultant units. Roman numerals added groupings (V, X, L, M, etc), making reading easier but introducing computational rules to operate on groups (e.g. V + V = X). The place notation of arabic numerals emphasizes readability, at the cost of introducing computational algorithms; for the first, two numbers sharing a space gave little indication of the resultant sum. Boundary numbers revert to very easy computation, at the cost of making reading a bit more difficult, since boundary numbers have many forms, one of which is minimal.

Depth notation corrects an undesirable characteristic of place notation, that of right-to-left linearity. The advantage of a base system is maintained, while

the dependence of the base index (i.e. 1, 2, 4, 8, 16,... in binary) on its sequential position is removed. The additional dimension used by boundary forms returns parallelism to numerical computation.

===
separate reading from calculation from meaning

Computer Implementation

Our Conventional Number Systems

When we think of a number, we imagine a series of digits strung together in a line, with the left-most digit representing the largest component of the number, and the right-most digit representing the units in the number. This representation of numbers, and in particular integers, has become accepted as universal. Our base-10 positional notation is the cumulative result of dozens of explorations of number made by every civilization of record. There are many ways to write a number, and our current system is considered to be by far the best.

Previous number systems have been defined by these features:

one-to-one mapping: units can be counted
special names for larger groups: a handful, a dozen, a score
the additive principle: combination of the parts represents the sum

Our current notation for numbers rests on:

one-to-one mapping: units can be counted
names for group magnitudes: powers of ten
uniform base: multiplication by a single base magnitude
functional zero: spaces are filled with symbols of nothing
place notation: dimension can be mapped to sequence

The current positional notation has evolved to ease reading, recording, and calculation of quantity. Conspicuous in its absence is the Additive Principle.

Types of Numbers

Any particular number can be written down in a variety of notations. Numbers are also used to mean a variety of things. The simple 1+1=2 is not so simple, as was emphasized by the Basic Arithmetic Sensibility Test. We seamlessly integrate several types of numbering systems into our daily lives. These different types of numbers include categories, unit ensembles, cyclic numbers, ordinal numbers, probabilities, and truth values.

One way to look at the varieties of number is to start with the familiar, and successively remove assumed structures. Another way is to begin with something very simple and successively add new structure. Either way results in the Hierarchy of Measure Structures in Fig %%%

==========

INDICATIVE: the elementary domain of perception. Where our mind is. This one is not taught in school. See Spencer-Brown, Laws of Form.

NOMINAL: a set, unordered collections of things. Eg: the space of library collections, of fish in a pond, of items on a menu.

ORDINAL: ordered, ranked things. Eg: your list of most to least favorite novels, pecking orders of fish, steps in an instruction.

INTERVAL: order in which the distance between items is equal. Integers. Eg: pages in a novel, age in days of fish, cells on graph paper.

RATIONAL: intervals which support ratios. Numerical fractions. Eg: percentages of each letter in a novel, portions of a meal each fish eats, comparison of monetary wealth.

REAL: continuous space. Real numbers. Eg: our model of the space underlying words on a page, the weight and length of fish, physical space.

IMAGINARY: contradictory spaces. Sqrt[-1]. Both True and False. Eg: our construction of mental images from words, wave propagation, inside a black hole.

Fig: %%% Hierarchy of Measure Structures
==========

[[discuss]]

Note how going down this hierarchy successively adds more mathematics and less physical reality. Note how cognitive spaces bound both the top and the bottom.

===
In reference to the nature of SPACE: The mathematical theory of measurement provides a concise summary of the generic types of spaces. Here they are, with slight elaboration. This list is a hierarchy, each following type is an elaboration of the preceding ones. Note that each can be conceptualized as one-dimensional, additional dimensions (2D, 3D, etc) are merely orthogonal products of more than one space. For grounding, it is commonly assumed that our everyday living space is composed of three REAL spaces at right angles.

In fact, this idea was made up in the middle of the sixteenth century by Descartes. TIME is just another space, one that we have forgotten how to travel freely in. So is SCALE.
===

Here is how each type of number works.

Indicative

iconic systems, usually not mentioned in the same breath as mathematics

Nominal (Categories)

Categories are the simplest type of numbering system, so simple that we do not even consider a set of categories to be numeric. Each week has exactly seven days, and each day has a special name. When one week has ended, we apply the same special name to a different day, and we generally manage to avoid confusion.

-- medieval numbers as memorable objects

Ordinal Numbers

--standing in line with a number

Interval Numbers

Rational Numbers

Cardinal Numbers (interval, rational, real??)

Our number system rests upon two impressive innovations that have been refined over literally thousands of years. The first innovation is the idea of a uniform base system; the second is an explicit zero.

Truth Values

-- truth or not

Unit Ensembles

The simplest case is unit form (also called stroke or tally arithmetic), a one-to-one mapping between integers (an abstraction) and a collection of atomic objects. The stars on an American flag represent the States the compose the

United States of America. Each State is a unique individual, and each is represented by a single star. The States are not counted on the American flag, rather their stars stand in concert, white upon a blue background, organized in rows (but no longer in columns), but not identified with a numeral. Instead, it is an option to map each star onto and unique but abstract integer, counting up in sequence from 1 to 50. The last number in the sequence is the count of the stars on the flag, and by reference, the number of States within the Union.

Stripes on the American Flag also represent States, this time the original 13 British colonies that banded together to establish the United States. Again, the stripes are not numbered, and there is no precedent that connects any particular stripe to any particular original State. There is also no precedent that connects a particular stripe to the star which may represent the same State.

Cyclic Numbers

We use cyclic numbers extensively when we describe time. A minute has sixty seconds, a day has twenty-four hours, a week has seven days.

Probabilities

--chances of rain

Real Numbers

Imaginary

segue: types and uses about modeling
        bases and positions about reading and writing

HUMANKIND'S GREAT ACHIEVEMENT

Our modern, and universal, base-10 place-value notation for integers has been glorified as one of humankind's greatest achievements [refs], as significant as the invention of the wheel.

===
quotes
===

All modern computational algorithms rest firmly upon single-base positional notation with magnitude implicitly identified by the sequential location of each digit. Digital computers universally use base-2 rather than base-10 since base-2 is easier translated into Boolean logic gates. Both the base-2 of computers and the base-10 of commerce share the idea that shifting a digit one place to the left multiplies its value by one order of magnitude. Fig %%% And this scheme requires the reputed coup de grace, an explicit zero to fill place locations that have no value.

==========
Fig %%% 1 10 100 1000...
==========

A theme of this monograph, that place-value notation can be improved upon both abstractly and computationally, might appear to be somewhat heretic. A critical analysis of positional notation requires reconsideration of several fundamental assumptions about the structure of arithmetic itself, specifically questioning

-- the linch-pin notion of an explicit zero,
-- the desirability of a strictly linear string notation for numerals,
-- the group theoretic relations such as commutativity and associativity
that permit rearrangement of token-strings on a line, and
-- the underpinnings of set theory as a necessity for numerics.

We share Kempe's goal: "...to separate the necessary matter of exact or mathematical thought from the accidental clothing -- geometrical, algebraical, logical, etc." ref[Kempe]

We do not introduce fundamental new theorems, neither do we discard existing mathematical tools and techniques; rather we suggest that the current structural constraints that define integer arithmetic can be liberalized, without loss of formality or efficiency. In particular, we demonstrate an advantage in removing the syntactic constraints imposed by notations based in the total ordering of strings and by relations that have a strict, usually binary, arity. We suggest that arithmetic can be formulated more naturally using parallel rather than sequential processes. That it can be formulated for understanding by using

icons, pictures, and diagrams rather than abstract token-strings. We ask why is it necessary to phrase something as common and useful as arithmetic in obscure and abstract languages designed for computers?

We begin by separating place-value notation from the concepts of counting, magnitude, one-to-one correspondence, and the addition and multiplication operations of arithmetic. The theory of arithmetic is independent of place-value notation; base-10 notation is a convenience and is not central to the ideas underlying number. We extend the separation of notation from concept by suggesting that properties of relations such as commutativity and associativity are also peripheral to the concept of number. For this reason, we focus on unit ensembles, the simplest possible representation of number. We treat rigorously the number systems conceived at the dawn of civilization, and used by preschool children to understand number. We suggest that arithmetic and algebra should be very simple, in use and in theory, and should be kept simple until a student become a mathematics major in college.

===
II.2.3 Hindu-Arabic Place-notation

Our Hindu-Arabic positional number system is string-based, it assigns a place-value to the location of a digit in a totally ordered sequence. A conventional integer is written as a polynomial of powers of a base; this choice impacts the complexity of basic algorithms for elementary operations and removes the visual and tactile intuitions and parallelism common of other, older numerical systems such as unit arithmetic. Thus, every place-value numeral, such as 2043, includes within it a significant amount of mathematical sophistication that requires thinking and interpretation on the part of the student.

Place-value notation exchanges convenience in reading for a modest inconvenience in computing. It is difficult to imagine a simpler computational operation than pushing piles of objects together, as in unit-addition. Addition of place-value numbers requires memorization of "digit addition facts" and the mechanisms of carrying and borrowing to manage place-value alignment.
===

Uniform Base Systems

===
how they work
===

Decimal numbers compromise nicely between size of representation and ease of computation:

1222 + 2088 = 3 2 2+8 2+8

In a standardized base (say 10), the positions in a string can be used to count the number of times the based is compounded with itself through multiplication. The number-name 20458 stands in place of

2x10^4 + 4x10^2 + 5x10^1 + 8x10^0

Positions are counted-off right-to-left, with the base-ten magnitude multiplying the dimension of the base:

number-name	2	0	4	5	8
magnitude	10000	1000	100	10	1
dimension	4	3	2	1	0
"place"	5	4	3	2	1

The positional technique forces the number 0 into each position for which the magnitude is 0, so the the number-name will not include spaces.

Place notation imposes a sequentially onto number-names; calculation algorithms must include techniques for interfacing adjacent places, called "carrying" and "borrowing".

Base-1

Base-2

Base-10

Decomposition Strategies

Unit form is the ultimate additive decomposition, while prime factorization is the ultimate multiplicative decomposition, each number represented by the product of its smallest prime factors.

Roman numerals use grouping convenient for finger computations. Arabic numerals use a uniform base with string position representing the dimension of the base.

representation	abstract	example of 204
unit	1+1+1+1+...	204 dots
Roman	I, V, X, L, C, M	CCIIII
base-10	ax10^n + bx10^n-1 +...+ nx10^0	204
base-2	ax2^n + bx2^n-1 +...+ nx2^0	110001100
prime factors	2^i + 3^j + 5^k + ...	2x2x3x17
boundary-2	(((i)j)...)k	(((((((•)•)))•)•))
boundary-10	(((i)j)...)k	((2))4

Explicit Zero

UNIT AND BOUNDARY ARITHMETICS -- SECTION II October 1, 2008 10:55 PM
William Bricken
January 2007

A Theory of Representation

We briefly examine three different representational systems for elementary mathematics:

-- unit-arithmetic (also called tally or stroke arithmetic),
-- the conventional Hindu-Arabic place-value notation, and
-- boundary arithmetic.

These three representational systems align with Bruner's (1966, pp. 10–11) three types of representation for mathematical operations: enactive, symbolic, and iconic.

Unit-arithmetic represents operations, such as addition and multiplication, as spatial combinations of sets of marks, maintaining a mapping between concrete activity and arithmetic operation. Place-value notation represents operations by collections of rules and algorithms to be memorized since they bear no resemblance to concrete operations. Boundary arithmetic represents operations concretely, as joining and substituting forms. Boundary arithmetic also incorporates a unique rule-based standardization process that is purely abstract, yet due to the spatial nature of boundary numbers, enjoys a resemblance to concrete operations. The standardization process achieves the efficiency of place-value notation without abandoning the naturalness of concrete operations.

Form or Content?

Kaput [ref 1987] is directly critical of the emphasis of form over content in elementary mathematics. He sees the predominance of math education addressing a particular set of representations and algorithms.

Mathematically, the operation of addition, for example, is an abstract concept that should not depend upon the particular embodiment of a representation. But certainly adding blocks by pushing piles together differs significantly from adding the cardinality of sets of blocks by following abstract symbolic rules that apply to disembodied symbols with meanings determined by convention.

Algorithms are known to depend upon representation. In fact, the discipline of Computer Science specializes in development of algorithms with particular mechanical efficiencies based on a variety of data structures. In digital computation, for example, decimal numbers are abandoned entirely in favor of binary numbers. In math education, the algorithms for manipulation of fractions differ from those of decimals, although fractions and decimals can express the same abstract magnitude. Operations on diagrams representing fractional

quantities are of a fundamentally different, concrete nature than the algorithms for both symbolic fractions and decimals.

Much of elementary mathematics dwells on specific algorithms for a singular specific system of representation (the decimal place-value system). This pedagogical choice is steeped in pragmatism. The decimal system is particularly convenient for reading large magnitudes, and for operations on multidigit numbers. The decimal system algorithms are certainly learnable, but for higher or more abstract mathematics, they may be islands of rigidity that interfere with progress in mathematical understanding.

Enactive Systems

Enactive systems are concrete; addition occurs by physically placing objects together. Although unit-arithmetic incorporates abstract strokes or marks, addition in unit-arithmetic is achieved by the action of placing marks together in the same space. This form of addition exemplifies the Additive Principle that guided historically early forms of non-symbolic arithmetic.

Symbolic Systems

Symbolic addition abstracts the cardinality of a set of objects into a symbolic name such as 3 or 7. The rules of combining these symbolic names guide the determination of their sum. Symbolic addition does not follow the Additive Principle; the symbols being added no longer possess the structural properties of the cardinality of the set they represent. As a consequence, we must memorize number facts and algorithms to determine a sum. Of course, symbolic numbers can be decomposed into units, reverting their additive behavior to that of unit-arithmetic.

Iconic Systems

Iconic addition is most commonly seen as pictures of groups of objects with specific cardinality. Since Bruner's writing, iconic systems have acquired a richer meaning, they can be dynamic and animated. Virtual manipulatives permit iconic forms to participate in computation. Boundary numbers provide a yet richer type of iconic representation that combines the abstract structure of symbolic forms with the manipulative behavior of enactive forms. Animation of boundary number operations shows the process of arithmetic.

Multiple Representations

Number systems support numerical abstraction; in turn, their abstract nature supports many concrete interpretations. "The usefulness of numerical ideas is enhanced when students encounter and use multiple representations for the same concept."[ref] [ref howmath p292] suggests that "Mathematics programs in the early grades should make extensive use of appropriate objects, diagrams, and other aids..."

Communication about numerical concepts requires an external representation. "Physical representations serve as tools for mathematical communication, thought, and calculation, allowing personal mathematical ideas to be externalized, shared, and preserved.12"

Addition, for example, is introduced in elementary school as the act of joining together collections. The cardinality of the combined set is the sum of the cardinalities of the sets prior to joining. Other representations, such as joining together structured base-10 (Dienes) blocks, or measured segments of a number line, are also used to illustrate the abstract act of addition. "Because many mathematical representations are suggestive of the corresponding metaphors, mathematical ideas are enhanced through multiple representations, which serve not merely as illustrations or pedagogical tricks but form a significant part of the mathematical content and serve as a source of mathematical reasoning." [ref]
===

How People Work

Piaget people
embodied vs abstract
digitality -- what it means to be human
history -- architecture w/o plans, learn via experience
egypt no algebra or abstraction
organic pragmatic vs binary
how brains are not computers
spatial intuitive interaction (not ed)

Math is Messy

Information Processing

Information processing models focus on our understanding of how people think. Using the model of automated deduction, Newell and Simon studied the way people go about unwinding symbolic puzzles @cite[humprobsolve]. Their work pioneered information processing models and the methodology of protocol analysis. The theory of Information Processing Systems (IPS) provides a detailed simulation of the dynamics of cognitive processes in individuals as they solve problems. Protocol analysis provides a methodology for studying individuals as they solve problems.

This appendix serves as an introduction to information processing models. It describes the theoretical context within which specific theories of error processes have developed, and outlines the constructs that constitute the information processing perspective. Davis' frame-based model is included as an example.

Overview of the Model

The information processing model proposes a task-specific model of cognitive processes. Errors can be associated with each step in the processing sequence of input, interpretation, transformation, processing, and output. These steps are simulated by a software program under executive control. Differences in hardware implementation and software style are ignored. The similarity between human and computer processing is assumed. The IPS perspective expands psychological modeling to include internal, cognitive processes that interact with information in the same manner as computers. To the extent that computation is algorithmic, the model is deterministic. Since the critical aspects of computation are related to input and output (that is, implementation dependent processes are not emphasized), the IPS model is neo-behavioristic. Although internal cognitive processes are modeled by computational processes internal to the computer, the model does not suggest that the implementation of the processes used by the automated system are similar to those used by humans.

The Information Processing Model

The human information processing model grew out of Newell and Simon's pioneering study of human problem solving @cite[humprobsolve]. Newell and Simon assume that people and computers are similiar in that both are information processing systems. The IPS theory has these broad characteristics @cite[humprobsolve ", Chapter 1"]:
@itemize[

@b[Process:] The theory assumes a set of complex cognitive processes or mechanisms to which behavior can be reduced.

@b[Content:] The theory is oriented toward content. A process model must be able to perform the tasks it explains.

@b[Sufficiency:] The focus is on mechanisms that are sufficient to perform and explain the task under investigation.

@b[Dynamic:] The theory addresses changes over time using the @i[program] as its formalism.

@b[Individual:] Processes are associated with individuals not with groups.

@b[Empirical:] Experimental design and statistical analysis are inappropriate because of the dynamic, history-dependent, high content topic of study.
]

An @i[information processing system] is "...a system consisting of a memory containing symbol structures, a processor, effectors, and receptors" (p. 20). Newell and Simon expand upon traditional mathematical description by including a process language that refers to changes in the state description of the system. "Symbols that designate expressions, elements of expressions, characteristics of expressions, or differences between expressions belong to the state language for the problem. Symbols that designate operators for transforming expressions, sequences of operators, or characteristics of operators belong to the process language" (p. 75). Problem solving consists of recognition of the initial and final states of a problem, and the application of permissible operators to states of the problem to transform the initial state into the final state.

Simon explains his early sketch of this model @cite[a&s ", p. 3"]:
@quotation[@blankspace[1 line]

"If we can construct an information processing system with rules of behavior that lead it to behave like the dynamic system we are trying to describe, then this system is a theory of the child at one stage of the development. Having described a particular stage by a program, we would then face the task of discovering what additional information processing mechanisms are needed to simulate developmental change -- the transition from one stage to the next. That is, we would need to discover how the system could modify its own structure. Thus, the theory would have two parts -- a program to describe performance at a particular stage and a learning program governing the transitions from stage to stage."

]

Although IPS was developed as a theory of human problem solving, the techniques and the model are strongly linked with computer science in general, and Artificial Intelligence in particular. An evaluation of the utility of this theory requires the specification of the relationship between the human behavior and the computational model. One extreme is to believe that the mind operates in the same manner as a computer. A softer position is to see the connection between mind and computer as a metaphor. The theoretical constructs of information science "... are assumed to @i[refer] to the contents, structures, and processes of the mind, but because the approach is new, the metaphorical nature of their reference shows through, as it did in Bohr's early solar model of the atom" @cite[jrme84-kaput ", p. 148"]. Specifically, the software metaphors incorporated in a program restrict our conceptualization of mental processes @foot[As a simple example, until a few years ago, almost all thinking about program design was anchored to the von Neumann model of a single, sequential processor. To impose sequential processing metaphors upon a cognitive model is to assume that the brain is like a sequential computer, which may be incorrect since the connection between neurons is massively parallel.].

IPS theories assume that software and implementation styles are inconsequential to the model. The theory incorporates constraint into models by being @i[task specific]. The environment, in the form of a specific problem to be solved, is held constant, while individual responses to the task are permitted to vary. The assumption is that a subject confines his responses to the specific task. In Newell and Simon's study of symbol manipulation within the domain of propositional calculus @cite[humprobsolve], consenting adult subjects cooperated with the researchers to describe their processes of solution. Preliminary training was a component of the experimental design, so that both researcher and subject felt they were working in the same domain.

Cognitive systems are assumed to be @i[adaptive], in the biological sense. "... adaptive devices shape themselves to the environment in which they are embedded" (p. 789). Cognitive processes are studied by observing their adaptation to a constant task environmnet.
@quotation[@blankspace[1 line]

"The shape of the theory we propose can be captured by four propositions:
@enumerate[

A few, and only a few, gross characteristics of the human IPS are invariant over task and problem solver.

These characteristics are sufficient to determine that a task environment is represented (in the IPS) as a problem space, and that problem solving takes place in a problem space.

The structure of the task environment determines the possible structures of the problem space.

The structure of the problem space determines the possible programs that can be used for problem solving" (p. 788).
]]

The central concept of the @i[problem space] refers to the entire set of potential actions for the subject, including imagined activities. Problem solving behavior is rational movement through the problem space. A measure of adequacy of the theory can be obtained by comparing actual behavior to behavior predicted by the goal seeking model of the IPS moving through problem space.

Since IPS is a cognitive theory, physical behavior is seen as part of the environment, while cognitive behavior is internal to the IPS. To appropriately constrain physical behavior, the task environment must be carefully selected to require a unidimensional ability. The source of variation across subjects for unidimensional tasks is assumed to be the difference in each subject's representation of the problem space. The complexity of this representation is constrained by the selection of naive subjects to study. Task skill is defined as the subject's ability to efficiently transverse the problem space.

The computational model offers a fundamentally new role for student errors. The Tyler model of curriculum, for example, views an incorrect response from the student as evidence that the lesson objective was not achieved, calling for either a change in teaching strategy or a change in evaluation techniques. The source of the error, the student's misconception, is not directly addressed at all. In the computational model, an error is positive information used to guide the modification of the algorithm that generated it. The intent of testing (evaluation) is to create errors which specifically identify weaknesses in the student's algorithm so that it can be revised to be more robust. The iterative development of knowledge follows this course:
@begin[enumerate, spread=0]

Inclusion of a small, modular skill increment,

Extensive testing of this increment, both independently and in the context of other knowledge,

Correction of discovered errors, and

Iterative generalization of the skill.
@end[enumerate]
The important point is that errors are the source of skill development. To paraphrase Papert: the question to ask about a student's knowledge is not whether it is right or wrong, but whether it is fixable @cite[mindstorm].

Models of Cognitive Processes

For modeling cognitive processes, IPS proposes the conceptual entities of structure, process, and information. They are composed to form the concept of a @i[mental model] @cite[cogsci81-johnson-laird, mentmod]. Mental models form the basis of the computational theory of cognition.

The components of an IPS follow. Each component supports a specific type of error.
@itemize[

@b[Input]: A parser transforms the external representation of the problem into an internal representation. Errors can arise when the parser does not know the grammar of the input or when the input to internal representation map is faulty.

@b[Interpretation]: An interpreter maps the parsed input into an internal context or @i[operative space]. Errors can arise when the interpreter maps the internal representation into an incorrect operative space or when the operative space is not furnished with appropriate tools for processing the internal representation.

@b[Transformation]: A set of transformation rules are applied to the internal representation within the operative space. Errors arise when the transformations are inadequate or incorrect.

@b[Processing]: A processing sequence allows the transformation rules to be applied to the input repeatedly. Errors arise whenever the processor fails to apply appropriate transformations.

@b[Output]: An output device conveys the result of transformation sequence as an external representation. Errors arise in the back-translation process.
]

Models of Error Processes

Suppes specifies the principle features of IPS models:
@quotation[@blankspace[1 line]

"The fundamental psychological assumption is that the student has an internal model of any skill he is using to perform a task. This internal model is responsible primarily for the errors generated.... The analysis of errors made by the student leads to insight into the bugs in the student's model of the procedures he is supposed to be applying" @cite[is-suppes ",p. 299"].
]

The computational model adopts the generative, or constructivist, position that a cognitive theory must generate the phenomena it addresses. A @i[generative model] achieves explanation by duplicating performance rather than by prediction. That behavior is generated by non-random algorithms indicates a return to determinism as a model. Suppes sees this determinism in the adherence of ITS workers to the belief that students do not act randomly.

IPS models finesse the problem of individual differences by addressing general competence rather than specific behavior. A @i[competence model] seeks to explain what a person @u[can] do in the context of a particular task, not what he does do. Thus, experimental observations of samples can be seen as information about individuals by the assumption that any of the responses seen in the sample could conceivably have come from any specific individual.

During the 1980s the fields of psychology, math education and computer science have all begun to adopt the IPS model.

Radatz provides an example of an error taxonomy that incorporates components representative of information processing models @cite[jrme79-radatz]: @foot[These categories are all susceptable to finer resolution. Pippig, in @cite[jrme79-radatz], for example, proposes a further classification for the rigidity in thinking category:
@begin[itemize, spread=0]

Perservation: single elements of the task dominate.

Association: incorrect interactions between single elements.

Interference: operations or concepts mutually interfer.

```
Assimilation:  incorrect input.

Negative transfer:  an erroneous impression from previous tasks.
@end[itemize]
]

@itemize[

Language difficulties:  misunderstanding the semantics of the representation.

Obtaining spatial information:  iconic representations fail to communicate
with visual students.

Deficient mastery of prerequisite skills, facts and concepts:  including
ignorance of algorithms, incorrect procedures, and insufficient
understanding of concepts.

Incorrect associations and rigidity of thinking:  negative transfer and
inflexability in approach.

Irrelevant rules or strategies:  application of comparable but incorrect
rules and overgeneralization.
]


Davis' Frame Model

The work of Robert Davis serves as an illustration of the application of
information processing theory to error processes in mathematics
@cite[cmb75-davis, mi8-davis, learnmath].  His book @u[Learning
Mathematics] provides a comprehensive example of the IPS perspective.

Central to Davis' IPS model is the @i[frame].  @foot[Thyne was probably
the first to introduce the concept of a frame into the modeling of error
processes @cite[patoferr].  He lacked, however, the computational
vocabulary to adequately conceptualize the processes that make frames
useful.]  A frame is a knowledge representation structure consisting of
a collection of attributes, or @i[slots].  Each collection of attributes
in a frame defines an abstract pattern, that, when matched or
@i[instantiated] by specific input, becomes active, producing an output.
For example, the @i[primary-grade addition frame] @cite[learnmath ", p.
113"] has two slots for input digits.  The frame unites specific digits
by addition to produce the sum as output.  It is rather strange that a
knowledge representation structure produces an output.  Davis' concept
of frame has an autonomous aspect: some control structure is constantly
seeking to activate the frame by finding instances.  Instances are
```

identified by visual cues and procedures such as pattern matching.
Activated frames generate behavior. If an active frame is lacking
specific instances, either a @i[default] instance, such as a stereotype,
is provided by the frame, or the frame fails to function, generating an
error. Davis sees the activity of a frame as question generating. When
sufficient answers are provided by the environmental input, the frame
creates output. Frame related failure can occur for several reasons,
including lack of the appropriate frame, a flawed or incomplete frame,
and failure to locate the needed frame.

Frames correspond to Piagetian @i[schemas]. They are the mechanism by
which large chunks of knowledge are coordinated into coherent wholes.
"Piaget's @i[assimilation] clearly corresponds to what we have called
@i[frame retrieval] and @i[frame instantiation]. Piaget's
@i[accomodation] corresponds to the synthesis of new knowledge
representation structures" (p. 158). Thus, frames "... serve as
`assimilation schemas' for organizing input data" (p. 125).

Davis lists some principles of his information processing model
(p. 108):
@begin[itemize, spread=0]

Discriminations are only as fine as necessary.

Representations are not actually deleted from memory.

The top-level program must run.
@end[itemize]
These constraints are combined with the assumption of @i[commonly-shared
frames] to provide a rough model of how errors are generated. Errors
are caused when input does not match the structure of the active frame.
Mis-matching input is resolved by modifying the input itself, or the
mapping from input to slots, rather than changing the structure of the
frame. Once created, frames are persistent. Frames are created
following orderly rules, such as the @i[rule of initial
over-generalization] and the principles listed above.

Davis acknowledges that his theories are provisional and eclectic. The
frame model lacks formal constraint on what can be made into a frame,
and has not clarified the mechanisms of how slots, or attributes, are
identified and clustered. However, "...the nagging problems of frame
instantiation, frame genesis and assembly, and frame coordination [are]
shared by all cognitive science" @cite[jrme84-kaput ", p. 151"].

===

Early Childhood Understanding

Elementary arithmetic education, in contrast to the conceptual abstraction of higher math, depends heavily upon direct experience with concrete objects. Number sense originates from visual and tactile interaction with objects during the preschool years, and is reinforced by concrete manipulatives in the early primary grades. Children commonly learn numerals, the one-to-one correspondence of counting, and the methods of grouping units into collections of a given size through the use of physical manipulatives such as attribute blocks, number rods, base-10 blocks, coins, balance beams, number lines, and fingers. Students are then asked to transfer their understanding of mathematical concepts from direct experience to the symbolic abstraction of string-based arithmetic and algebra. The mathematics curriculum then rapidly becomes symbolic and abstract, obscuring understanding with unnecessary formal structure. Mathematical concepts first instantiated by spatial and tactile interaction with objects must be converted into rigorous algebraic manipulation of abstract, and somewhat arbitrary, strings of tokens, creating a severe discontinuity in both the style and the content of elementary mathematics education. We seek a form of arithmetic that unites formal abstraction with direct experience.

As Easy As One, Two, Three

K-2 7-9 gap comes form "now put it down on paper"

Tension Between Abstract and Concrete

[ref howmath] points to an inherent tension between abstract and concrete aspects of mathematics "This tension is a fundamental and unavoidable challenge for school mathematics."

Essentially, a student is expected to first learn using concrete applications of abstract and very general numerical concepts, and then to transfer this learning into the abstract structures of group theory presented as the Rules of Symbolic Algebra. "That kind of learning often takes time and can be quite difficult." [ref]

[ref howmath] recommends that "Different ways of representing numbers, when to use a specific representation, and how to translate from one representation to another should be included in the curriculum." At the same time, a failure to adequately use geometrical structures, especially the number line, as representations of numerical concepts is acknowledged.

We propose to develop and explore a new type of spatial representation that maintains a formal mapping to symbolic structures while at the same time presents many of the advantages of direct manipulation. More fundamentally, we intend to create curriculum materials that provide the opportunity to explore some representational inconsistencies between symbolic and non-symbolic (concrete, visual, manipulative, and virtual) math teaching tools, and to formatively evaluate the contribution to mathematics understanding (and misunderstanding) made by each. The essential problem is that common concrete models of numeric operations do not map directly onto group theoretic concepts. To illustrate this tension, we now briefly examine two common representations of elementary numerics, unit-arithmetic and place-value arithmetic. These models lay the groundwork for describing boundary-arithmetic, a representational system that combines the intuitive operations of unit-arithmetic with the convenience of place-value notations.

Two Early Studies Define the Research Issues

Two early studies of errors @cite[est13-phelps, est17-smith] provide a summary of error phenomena and of the methodological difficulties in studying errors.

Phelps analyzed errors in addition facts @cite[est13-phelps] @foot[Data are from 238 eighth grade students of eight schools in San Jose, California. Each student answered a one-minute long, single-digit addition test, administered twenty-five times (five times a day over five successive days). 5950 tests form the data set; each test consisted of the 55 combinations of single digit binary additions.]. His motive was to provide diagnostic information about how often errors are made, how many students make them, and which number facts are most difficult.

Phelps found extensive variation in errors, across problems, students, and schools. He found that samples from different schools varied widely in their error behavior, from a low of 16 errors per student to a high of 70 errors per student on the same battery of arithmetic questions. There were also regularities in Phelp's error data. Combinations with large single-digit integers (such as @math[9 + 7]) were more difficult than those with smaller digits. This result was unexpected because the sample of eighth grade students supposedly received equal practice on all combinations. Certain combinations, like adding one (@math[9 + 1]) and adding double numbers (@math[3 + 3]), exhibited specific @i[pattern] errors, out of proportion to the expected error distribution based on equal difficulty. These errors seem to be associated with the syntax of the problem, and not with its meaning. Although there were very few errors on @math[4 + 4] or on @math[7 + 7], there were many errors on @math[3 + 3]. Wrong answers were attributed to an apparent confusion

between addition and multiplication, since 90% of the incorrect answers were the number @math[9]. Phelps was unable to explain why this error occurred frequently. The @math[9 + 3] combination presented another problem: of the 304 errors on this problem 70% were the number @math[11], while only 12% were the number @math[13]. Phelps expected an equal likelihood of being one over or one under the correct answer.

The Newman Hierarchy

Newman's hierarchy of the causes of errors is based on an information processing model @cite[resinmathedaust-newman]. It posits a step-wise sequence of skills:

@itemize[

@b[Reading:] symbol recognition.

@b[Comprehension:] understanding the question and the symbols used.

@b[Transformation:] selection of appropriate processes to answer the question.

@b[Process skills:] performing the necessary operations. Subcategories include wrong operation, faulty algorithm, faulty computation, and random response.

@b[Encoding:] writing the answer acceptably.
]

The Newman Hierarchy has strong advantages: it has a consistent focus on student behavior, is useful for diagnosis by proposing an order in the occurrence of errors, and is well studied (Table @ref[newmantable]). It is weak in that it is not fine grained enough to track detailed mathematical processes, has only intuitive validation, and permits multiple classification of the same event. The Australian data indicate that about one-quarter of the population of errors is attributable each to selection of a process, applying the process, carelessness, and input/output.

@include[t2-newmanheirarchy.mss]

Individual variation over time, when it was examined, provides a new perspective on carelessness. Watson observes: "... the children varied their way of working out a problem. A child would do one division sum one way, another in a different way, or he would change his mind half way through working out a problem" @cite[esm80-watson ", p. 327"].

Watson suggests that carelessness was an overassumptive category in that singular errors ae not all explained by carelessness. A student may generate a correct answer several times, and still revert to an incorrect algorithm without being careless.

A wrong answer in number work is apparently quite as definite as a right answer" (p. 21).@foot[
Within the rote learning pedagogy of the times, Myers concluded "Do not let him have a chance to answer wrong and do not let his classmates hear a wrong answer" (p. 24). This sentiment is echoed by Robert Gagne @cite[jrme83-gagne ", p. 15"]: "Teachers would best ignore the incorrect performances and set about as directly as possible teaching the rules for correct ones."]
]

Gibb

Gibb provides an example of excellent statistical methodology applied to children's thinking processes in subtraction @cite[jee56-gibb] @foot[Data are from 36 2nd grade students in 24 different schools in Wisconsin. The content was single integer subtraction.]. She designed a mixed model factorial study with two systematic independent variables, Application and Context, and one random variable, Individuals. @i[Application] refers to the type of subtraction situation (take-away, additive, and comparative) @foot[Subtraction can be used to solve different types of problems. @i[Take-away] problems present a group from which a part is to be taken. @i[Additive] problems present a part to which another part must be added to form a group. @i[Comparative] problems present two groups whose difference is to be determined.]. @i[Context] refers to the mode of representation of the problem (concrete, semi-concrete, and abstract) @foot[Arithmetic can be applied at different levels of abstraction. @i[Concrete] problems are presented in the context of manipulatives such as toys. @i[Semi-concrete] problems are presented in the context of geometric figures. @i[Abstract] problems are presented in the context of symbols.]. Each child was presented with all nine application by context conditions. Order of testing, and specific problems in each condition were randomized. Responses were measured on six dependent variables associated with performance (sophistication of process, degree of understanding, time to completion, and others). In the analysis of variance, Gibb found that the random effect of Individuals interacted with both Context and Application.

Gibb also tape recorded her interviews and testing of each student. Thus, she was able to examine protocol data to discover characteristic

patterns in behavior. Portions of her conclusions follow (p. 78):
@quotation[@blankspace[1 line]

"These children did not respond equally well to problems presented in similar contexts.

... there is no context in which problems appeared the same to all.

... in no case did all applications evoke the same behavior from the same child.

... one score does not give a uniform picture of the child's behavior in problem solving situations. Not only did different children bring many varied procedures to problem solving situations, but also the same child used many varied procedures.

... this study gave no evidence of a general subtraction ability which characterizes children."
]
Again, variability pervaded the data.

Ginsberg

To Ginsberg, the ease with which students solve problems is directly related to their knowledge of an appropriate and proven algorithm. Children rely on informal and invented methods when the algorithms they are taught somehow do not work. Because these informal methods return a credible answer more often than the unlearned formal algorithms, they are persistent. Particularly, it is often easier to solve a problem mentally than it is to structure, write, and solve it mechanically using symbols. Many errors are purely within the syntactic domain of written responses, hiding a student's existing understanding of how to solve a problem. These syntactic errors appear to undermine a student's belief in their semantic understanding, resulting in highly variable and error prone test behavior. Ginsberg explains that children first learn intuitive methods. Later, when introduced to written methods, they fail to understand, leaving a gap between understanding and algorithm. Ginsberg sees this gap as the source of exotic written errors. The point of failure is the mapping between external representation and internal capabilities.

Ginsberg summarizes his model of learning in this way (p. 129):
@itemize[

Errors result from organized strategies and rules. (These strategies

are not necessarily variants of correct procedures.)

The faulty rules underlying errors have sensible origins. (The origins are semantic as well as syntactic.)

Too often children see arithmetic as an activity isolated from their ordinary concerns. (This does not make arithmetic a meaningless activity; it does make it low in semantic content, and thus more susceptable to semantic error.)

Children demonstrate a gap between informal and formal knowledge. (This lack of connection generates errors.)

Children often possess unsuspected strengths. (Syntactic errors hide semantic competence.)
]

Finally, Ginsberg notes: "Individual children can display extraordinary inconsistency in mathematical behavior" (p. 128). One source of this inconsistency is that students widely @i[invent] mathematical techniques @cite[psymathinst].

Ginsberg's contribution is the explicit recognition of the semantic needs of novices. A limited notion of context dependency was introduced by Smith when he observed that the operators a student selects for a problem can be influenced by the operators required in previous problems. That errors interact with context has been observed by other researchers. This interaction is both within the substeps of a problem and across separate problems. In their study of temporary lapses, Knight and Ford observed different errors depending solely upon the position of digits in a problem @cite[esj31-knight] @foot[Data are from 200 6th grade students from four schools, on multiple-digit multiplication problems.]. Thyne observed errors that changed depending upon the problem that preceded the incorrect one @cite[patoferr] @foot[Data are from 1325 Scottish children in elementary grades, on arithmetic addition and subtraction facts.]. The key to understanding these results is provided by Ginsberg. The meaning that a student ascribes to a computational problem often overrides the syntactic structure of the problem itself. This semantic component interjects extra-domain information into the interpretation of symbols. Errors appear to be erratic because the relevant aspects of a student's global context can change often.

Elementary Algebra

The domain of elementary algebra is constrained to the solution, or simplification, of linear equations in one unknown. An equation is @i[elementary] if it does not require parentheses to indicate nested complexity. @foot[This implies one binary operator on each side of the equation.] Algebra errors can be conceptualized in much the same way as elementary arithmetic errors. The student is taught a simplification algorithm; he makes errors while applying the algorithm.

Wattawa conducted one of the first error analysis studies of algebra skills @cite[mt27-wattawa]. Her conclusion after recording the errors made by an algebra class over a period of three months is that
@quotation[@blankspace[1 line]

"... difficulties ordinarily attributed to algebra are in reality difficulties of a more deep-rooted nature. It would apparently seem as if certain fundamental processes, particularly those of arithmetic, have not become automatic enough for the poorer students to devote their entire attention to whatever is new to them in algebra" (p. 212).
]

Clinical interview data on algebra problems has been collected for many high school students @cite[cmb75-davis, focus81-burnett, basalg]. Almost all current research interest in algebra errors is directed toward the development of an information processing model. The automated simplification of complex algebra problems has been a research focus in Artificial Intelligence since the symbolic integration programs of the 1960s. IPS researchers have a well established vocabulary for the representation and solution of algebraic equations @cite[ai81-bundy, ijcai81-sleeman,modmathreas].

However, an observation in
@cite[cmb75-davis] brings the appropriateness of the IPS model into question. Davis notes that the mathematical demands of a problem are different from the cognitive demands. Specifically, mathematical tokens are used in several different ways; the interpretation placed upon them varies with context. For example, compound tokens such as @math[3x] represent both objects and processes. The student must learn to see compound tokens in several different ways. Davis' concern is reminiscent of Spencer's dissertation conclusion: ambiguities in the meaning assigned to tokens leads to most errors. This ambiguity is particularly confusing when variables are introduced.

The space of algebra errors is particularly difficult to map. Arithmetic skills are confounded with algebraic skills, many algorithms exist for the solution of algebraic equations, and the concepts

particular to algebra, the variable and the equality symbol, are subtle @cite[cpi-kaput, esm81-kieran, csms]. Bernard summarizes the research @cite[pme6-bernard ", p. 145"]:
@quotation[@blankspace[1 line]

"Indications are clear that the larger the number of steps, the less likely students are to correctly solve the equation. Possible difficulties are with combining like terms, transposing terms, doing computation, or in understanding fundamental concepts such as 'variable' or 'equation'."
]

In a study of algebra learning in novices, Greeno collected verbal protocol data @cite[pitts85-greeno]. He observed that novice performance was "...profoundly disorganized and fragmentary" (p. 3). Most errors were characterised as unsystematic. Greeno concluded that "...students' processes of representing expressions do not provide them with well-formed representations of the structural features of the expressions during their early stages of learning" (p. 11) @foot[This study is consistent with the results presented in this thesis.].
===

TEACHING MATHEMATICS

Something Is Not Working

Spatial Math Is Manipulable

We hope to provide curriculum tools and materials that address a significant discontinuity in the teaching of mathematics, the conversion students must face going from an elementary curriculum grounded in concrete manipulables to a secondary curriculum grounded in abstract symbols. This discontinuity is deeply connected to how we represent mathematical concepts, and is rarely brought into question. Only recently, with the advent of web-based virtual manipulables, has the display technology become available to promote diagrammatic and visceral mathematical systems from a second-class role as informal aids to a first-class role as rigorous formal tools.

It is well known that America's students are underperforming in mathematics education. On the Washington [State] Assessment of Student Learning (WASL) in 2006, for example, half of the students in grades 6 through 10 failed to meet grade-level standards of performance for mathematics [ref]. The 2006 overall current failure rate of 50% incorporates a 75% failure rate for Washington State minorities and a 70% failure rate for students living at the poverty level, a group composed primarily of white and Asian students [Shaw ref]. The WASL is based on the Principles and Standards for School Mathematics developed by the National Council of Teachers of Mathematics (NCTM) [Trafton ref].

After a decade of extensive effort, the 2006 results represent a significant improvement (!), having increased from a 33% pass rate in 1999. However, the continuing massive failure rate suggests that an exploration of innovative techniques in mathematics education may prove profitable.

Manipulables

Historically, at the turn of the twentieth century Frege and Hilbert exerted their considerable influence to exclude diagrams and "intuition" from mathematics and consequently from the mathematics curriculum [%ref]. The success of the entirely symbolic approach of Whitehead and Russell's Principia Mathematica, followed closely by the rise of the use of symbolic techniques in digital computers, has led to the expression of higher mathematics almost exclusively in symbolic languages. Elementary school teachers however, use manipulative and diagrammatic techniques widely. Due to their physical limitations, physical manipulables have been constrained to "concrete examples", just as drawing a triangle in geometry is a representation of a particular

triangle. However, with new software tools such as Geometer's Apprentice [ref], a student can "draw" a generic triangle, and use the diagram itself as an object of computation.

Poor math performance across America suggests that exploring innovative techniques may be beneficial. One potential source of math confusion is the difference in the mathematical approaches of concrete manipulables versus symbolic abstractions. Manipulables, for example, generally achieve addition based on the Additive Principle: the representation of combined parts is the representation of the sum. Symbolic addition is achieved by rule, the combined representations of the parts do not represent the sum.

Web-based diagrammatic and manipulative tools further confound the distinction between concrete application and symbolic abstraction. Their use is in direct opposition to the almost complete exclusion of diagrammatic, visual, and intuitive representations from formal symbolic mathematics.

===
fix
===
What is missing to date is a rigorous diagrammatic system of numerics, a system that is inherently interactive and manipulable while at the same time meets all criteria of symbolic formality. Boundary mathematics is such a system.

We do not propose to have a solution to bridging the gap between the experiential mathematics of elementary school and the algebraic abstractions of high school. We have, however, developed diagrammatic formal systems that are unexplored as tools for teaching. Our research proposal is to

-- construct curriculum tools in boundary mathematics
-- examine student performance in light of differences between
   manipulative and symbolic formal mathematics
-- analyze mathematical errors characteristic to each of these modes
-- refine the boundary math curriculum tools based on student feedback

Virtual Manipulables

One direction of growth in tools for teaching mathematics is the use of web-based virtual manipulables [refs]. A virtual manipulative is "an interactive, Web-based visual representation of a dynamic object that presents opportunities for constructing mathematical knowledge" ("What are Virtual Manipulatives?," Patricia Moyer, Johnna Bolyard, and Mark Spikell (2002) p. 373).

Virtual manipulables provide iconic models that simulate concrete manipulation. They currently bare a strong resemblance to concrete manipulatives such as

Cuisenaire rods, base-ten blocks, pattern blocks, rulers, number lines, logic blocks, fraction pieces, and geoboards. Virtual manipulables are decidedly constructivist. Students construct meaning by using computer input devices, such as the keyboard, mouse, touch-screen, joystick, etc., to control apparently physical actions of virtual objects through translation, rotation, flipping and other generally spatial transformations. For example, graphing linear equations can be made interactive both by generating a graph given student input of a linear equation, and by generating a linear equation that corresponds to student dragging and rotating the line graph itself [refs].

Physical (concrete) manipulatives have a long history in the classroom. Today, Web-based applets that provide virtual models of mathematical concepts and computations are burgeoning. The math archives [ref], for example, provides pointers to literally hundreds of tools, while the National Library of Virtual Manipulatives [ref] Interactive Math project at Utah State includes hundreds more. Principles and Standards for School Mathematics Electronic Examples--interactive applets for investigating concepts from NCTM [ref].

Many of these new electronic tools for mathematics learning provide a much broader experiential basis, they are manipulable models of abstract, symbolic mathematical concepts and algorithms. For example, abstract mathematical concepts such as cardinality are mapped directly onto spatial and visual analogs, such as collections of abstract identical objects.

Reimer [ref reimer], suggests potential benefits of virtual manipulatives for learning:

"Students liked the immediate feedback they received from the applets, the virtual manipulatives were easier and faster to use than paper-and-pencil, and they provided enjoyment for learning mathematics. Their use enabled all students, from those with lesser ability to those of greatest ability, to remain engaged with the content, thus providing for differentiated instruction."

In Bricken [1992] and Winn and Bricken [1992] we mapped abstract algebraic concepts onto properties of space. Addition was represented as sharing the same space without touching, while multiplication was represented by touching stacks of algebraic objects (constants, variables and signs). The key idea was that these mappings were dynamic: spatial manipulation achieved algebraic computation. Figure 1 illustrates this spatial algebra.

Spatial Analogs

Visual and spatial analogs and models of mathematical concepts are elevated to a curriculum design principle in How [ref howmath p292]:

"Design Principle 3: Providing Visual and Spatial Analogs of Number Representations That Children Can Actively Explore in a Hands-On Fashion"

Spatial analogs are lines, scales, dials, etc. that support the concept of number within their geometry. "Most students benefit from visual representations of concepts." [ref and [Fuson, 1986]].

The proposed tools based on boundary mathematics are not analogs, they are pure numeric forms that do not rely on an underlying geometric metric to construct an analogical relation to number. The forms of boundary mathematics are simply spatial numbers, in the same way that the cardinality of a given set is spatially represented by the members of the set.

Often though, multiplication is inconvenient within spatial systems. Dials and scales simply do not support multiplication. Cuisenaire blocks achieve multiplication via an increase in dimension, constructing an array that must be counted. Thus the additive principle is called indirectly, since a x-by-y array must be counted unit-by-unit to arrive at the multiplicative result. This provides a visual but not an algorithmic understanding.

===
Virtual reality is a computer generated, multi-dimensional, inclusive environment which can be accepted by a participant as cognitively valid [6]. VR teaching systems overcome the inconvenience of an insufficiently abstract physical reality by combining mathematical abstraction with the intuition of natural behavior. The programmability of VR allows a curriculum designer to embed pedagogical strategies into the behavior of virtual objects which represent mathematical structures [2]. Using a VR presentation system, the axioms of algebra can be, so to speak, built into the behavior of the world.
===

===to integrate

The difficulties children have when they begin to learn algebra are well documented [9] [7] [17] [8] [4].
Spatial representations enhance understanding [11]. Concrete manipulation is known to be an effective teaching technique [15] [1] [14].

===
[1] Berman, B., & Friederwitzer, F. (1989) Algebra can be elementary ... When it's concrete Arithmetic Teacher, 36 (8), 21-24.

[4] Bricken, W. (1987) Analyzing errors in elementary mathematics Doctoral dissertation, School of Education, Stanford University.

[7] Gerace, W.J., & Mestre, J.P. (1982) The learning of algebra by 9th. graders: Research findings relevant to teacher training & classroom practice Final Report, National Institutes of Health, Washington DC, (Contract # 400-81-0027).

[8] Greeno, J. (1985) Investigations of a cognitive skill Technical Report, Pittsburgh: University of Pittsburgh Learning and Development Center.

[9] Kaput, J.J. (1978) Mathematics and learning: roots of epistemological status In J. Lochhead & J. Clements (Eds.), Cognitive process instruction Philadelphia, PA: Franklin Institute Press.

[11] Larkin, J.H., & Simon, H.A. (1987) Why a diagram is (sometimes) worth ten thousand words Cognitive Science, 11, 65-99.

[14] Shumway, R.J. (1989) Solving equations today School Science and Mathematics, 89, 208-219.

[15] Sowell, E.J. (1989) Effects of manipulative material in mathematics instruction Journal of Research in Mathematics Education, 20, 498-505.

[17] Thwaites, G.N. (1982) Why do children find algebra difficult? Mathematics in school, 11(4), 16-19.
====

UNIT AND BOUNDARY ARITHMETICS -- SECTION IIB October 1, 2008 10:55 PM
William Bricken
January 2007

SPATIAL FORMS
- Spatial Formalism
  - Trees and Graphs
  - Diagrammatic Logic
- Featureless Space
  - A Simple Example
  - Both Inside and Outside
- Spatial Algebra

SPATIAL REPRESENTATION of ELEMENTARY ALGEBRA
- Introduction
- Spatial Algebra
- Group Structure of Spatial Forms
  - Commutativity
  - Associativity
  - Distribution
  - Identities
  - Additive Inverse
  - Calculus of Signs
  - Multiplicative Inverse
- Factoring
- Caveats
- Principles of Spatial Mathematics
  - General Principles
  - Mathematical Principles

SPATIAL FORMS

Formalism is at the heart of mathematics, it is the desire to have every concept and every representation rigorously defined, so that no possible doubt or ambiguity can arise. The sequential notation of token-strings has accompanied the rise of formalism. More properly, the development of formalism was accompanied by the rigorous removal of all other notational forms, including both unit ensembles and diagrams. We revisit formalism with a desire to incorporate intuitive models such as diagrams. Proper treatment of the concepts of spatial representation shows that token-strings are not the only representation that is blessed with rigor. A consequence of formalizing spatial forms is that we find an array of new, relatively unexplored concepts that can serve as a basis for arithmetic. These unexplored concepts were mostly present at the dawn of civilization about 8,000 years ago.

Spatial Formalism

We seek to develop a formal system that can unite the axiomatics and the pragmatics of arithmetic. The content that follows incorporates formal concepts that are anathema to many professional mathematicians [refs].

"...despite the obvious importance of visual images in human cognitive activities, visual representation remains a second-class citizen in both the theory and practice of mathematics." [ref barwise1991]

Specifically, we develop a spatial formalism not based in string notation. The notation of boundary arithmetic is not an informal convenience built on top of string-based expressions, it is instead a collection of formal techniques that instill upon spatial forms the same rigor as has been established for conventional token-string notation.

Trees and Graphs

Algebraic formulae are often treated as trees, and even graphs, suggesting that mathematical equations can be read as two-dimensional forms. But what are the specific properties that distinguish two-dimensional diagrams from one-dimensional strings? For example, the form x+y can indeed be viewed as a tree, as shown in Fig %%%. However, whenever the branches of this tree are assigned a "right" and "left" position, the tree representation is constrained to be equivalent to a totally-ordered string representation. A spatial form located in a featureless space has no anchors with which to associate the handedness of its branches. The primary topological property of the tree representation is that there is a distinguished node, the center one, that has multiple links connecting leaves. Each leaf, by definition, has only one link. However, when

different labels are ascribed to either link, the simple graph form becomes oriented in the plane. Topologically indistinguishable graphs become distinguishable by their handedness. This orientation grounds our perspective so that we cannot engage in spatial transformations (such as rotating 180 degrees) that otherwise would be freely available. We consider handedness in graphs, counting the number of links, labeling links, and in general construction of any privileged perspective, to be artifacts rather than structural features of a spatial display.

==========
Figure %%%: x+y as string tree spatial topo
==========

Similarly, the idea of specifying the number of links connected to a central node is extra-topological, thus imparting unintended semantics onto a purely spatial structure. Thus, for example, a binary tree incorporates a structural metric, that of a specific number of links, that is not intrinsically part of the spatial structure.

Diagrammatic Logic

Both diagrams and token-strings can represent what is meant by propositional logic. Frege and Peirce each developed a type of diagrammatic logic, a formal method of manipulating spatial structures to achieve logical deduction. Their innovation is that relations between spatial shapes can stand in place of strings of tokens.

"...there has until quite recently been a long-standing prejudice against non-symbolic representation in logic, mathematics, and computer science." [ref shin iconic p21]

The question to date has been whether or not diagrams can convey the same formal information as strings. Venn developed his diagrams with set theory in mind; Peirce developed his existential graphs with logic in mind; Hasse developed his diagrams with partial orderings in mind. But there has been a history of exclusion of diagrams from the formal foundations of mathematics [refs], and a general disregard for the attempts by Venn, Peirce, Frege, and others to incorporate the expressive power of spatial forms into formal mathematics. Since the diagrammatic approach incorporates types of structure that are simply not available within a string-based notation, we consider spatial formalism to be an asset rather than a liability.

Peirce's Alpha Existential Graphs are a boundary logic within which the formal structure of sentential logic is expressed by nested and juxtaposed circular boundaries drawn on a plane surface. Fig %%%. The semantics of logic is

embedded within the containment relations of spatial boundaries. In Laws of Form, Spencer Brown clarified Peirce's Existential Graphs by incorporated void-equivalence into an algebra that stands independent of logic, but has an interpretation as sentential logic. Fig %%% Boundary mathematics is built upon a new type of non-numeric imaginary first articulated by Peirce: void-equivalent form in a featureless space.

==========
Fig: AEGs
==========

==========
Fig: LoF
==========

Featureless Space

We begin with a featureless space, and the implicit or explicit boundary that differentiates that space from the surrounding context. Fig %%% We are thinking quite physically rather than abstractly, of the space surrounded by a glass or a grocery bag, the inside and outside of homes and cars, and in particular, the edges of the page these words are recorded upon.

==========
Figure %%%:
==========

We want to maintain certain features of space within the syntax of the mathematical systems we are considering. These features can be contrasted to aspects of the typographical space of a line, the words and letters arranged thereupon, and the rules of transformation that permit rearrangement of words, letters, and mathematical tokens. String notation, the totally ordered arrangement of tokens endowed with a semantics, is supported by a one-dimensional linear space, together with a collection of conventions about how to lay out long strings into lines on a page and pages in a book. Modern algebra rests upon rules of syntactic string transformation, with its semantics reflecting the concepts of the spatial arrangement available to ordered strings.

For example, the concept of commutativity is represented by two strings that are connected by a sign of equivalence. The circled dot, ⊙ , represents any commutative operator. The small letters stand in place of arbitrary objects.

$$a \odot b = b \odot a$$

or

$$Rxy = Ryx$$

Commutativity permits an arbitrary form to reside either first or second in an ordering of two tokens. It asserts that position on either side of the operator token ⊙ doesn't matter. Alternatively, we may say that the commutative relation supports rotation of argument tokens through 180 degrees. Fig %%%.

==========
Figure %%%: rotation
==========

The syntax gives permission to exchange the relative location of two universal variables, the semantics identifies a group theoretic symmetry. Alternatively, arguments could be labeled by ordinals, freeing them in space by constraining them by ordinal identification,

$$R(\text{first}=x, \text{second}=y) = R(\text{second}=x, \text{first}=y)$$

Ordinal labeling removes the ambiguity introduced by placing arguments in space by endowing each argument with an additional property.

When specifying the commutative rule governing the unit of a system,

$$1 \odot a = a \odot 1$$

we conventionally identify a "right" and "left" unit rule, mirroring the positional and ordinal perspective on arguments.

Now imagine providing a planar space for arguments,

==========
Figure (a b) = (b a)
==========

Here the relative location of the two variables requires additional information; in addition to left/right, we must also specify the meaning of up/down. From a Cartesian perspective, we have additional types of symmetry. The spatial freedom permitted to an argument token can encode a more complicated object. An example is an imaginary value on the complex plane, for which real numbers are arranged in a line from left to right while imaginary numbers are arranged bottom to top.

Here we examine the freedoms provided by not interpreting relative position, by not attributing arguments with additional properties. In a spatial vocabulary, two forms in space can be free to be anywhere within that space (although leaving the space for a different space would considered to be a violation of intention). We propose to remove the concept of distinguished arguments (as enforced by ordinal labeling for instance) from the operations of addition and multiplication.

If we consider the entire space enclosed by a frame as a single context, with argument tokens distributed anywhere within the frame, then commutativity becomes an irrelevant concept. Here we are interested in spaces that operate upon whatever they contain, regardless of relative position or rotation of forms within that space. The relative location of members of a set does not mater; similarly the relative location of spatial variables to matter. The primary difference is that the spatial representation shows the absence of ordering for members of a set more explicitly, by having space to put them into. The token-string representation cannot explicitly represent the positional relation between members because strings are organized in a strict left-to-right order, {a,b,c,...}.

The freedom of spatial position can be interpreted as super-commutativity, or one might say that commutativity is implicit within the spatial relation. The perspective presented in this monograph is that the concept of commutativity is irrelevant to a spatial relation, the property is neither confirmed nor denied, neither explicitly nor implicitly. We can elect to add either explicit commutativity or explicit non-commutativity as it becomes relevant, such as the case in which spatial structures are interpreted as non-commutative groups.

We will also develop the idea that spatial representation supports parallel network computational models, while string representation supports sequential, iterative computation. Our point is that axioms for a theory of integers can be constructed in both sequential and parallel notations. Our goal is to introduce a simple formal system that incorporates intuitive spatial forms and operations, that achieves the algorithmic efficiency of place-value notations, that is compatible with existing notations, and that is as expressive as modern algebra while avoiding the linear symbolic abstraction that token-strings impose upon mathematical understanding.

A Simple Example

Consider a collection of pennies placed on a table top, as shown in Fig %%%.

==========
Figure %%%: Pennies on a Table
==========

Every penny is physically distinct from every other penny, although each has the same value. The coins can be freely moved about the table top and stacked upon one another. The spaces occupied by each coin are disjoint, both pairwise and as an ensemble. There is no coordinate system that permits identification of a coin's position relative to an origin. No particular coin has a privileged position. In fact the only relations available to distinguish the relative position of coins are those of sharing the same table top, and of being stacked upon one other. The number of pennies on the table is arbitrary, but we can clearly distinguish base cases of no coins and of a single coin.

In this example, the structural relations between coins is a model of a boundary arithmetic. The space occupied by the pennies is not a three-dimensional Cartesian space, nor a metric space, nor an abstract dimensional manifold; rather, it is featureless. Even the flatness of the table is irrelevant to this model. All that identifies a "table-top" space are the edges of the table-top, and the presence of coins in the space defined by the edges. Strictly, the space containing coins and the configuration of coins themselves are co-defined.

Define addition as stacking; coins are added together by sharing a common stack. Every coin is itself a stack, so that there is no difference between a single coin and a stack of one coin. Here, the concept of addition embodies the Additive Principle: the representation of a sum is the representation of the parts constituting the sum. Fig %%%

==========
Figure %%%: Naive Semantics of Adding Pennies on a Table
==========

This model of addition is substantively different than that of symbolic arithmetic. Commutativity of the addition operation could be interpreted as an invariance of stack value under stack construction. That is, two stacks can be combined in either order without changing the number of coins in the resultant stack. Alternatively, commutativity could be interpreted as a permission to stack coins in any order. However, the spatial perspective is stronger still. The positional freedom of a spatial configuration of coins makes assembly order irrelevant. Coins are grouped within a space by being put into that space. There is no notion of which group goes first, nor is there a privileged group of coins that is added to. In contrast to a necessity for formal permission to

rearrange the ordering of a sum as in string notation, in the spatial model, ordering is semantically irrelevant and syntactically inaccessible. Stacks may be combined concurrently, and multiple stacks may be combined at the same time, precisely because the spatial configuration does not distinguish the concepts of spatial and temporal ordering. For spatial semantics, the concepts of commutativity and associativity of addition simply add structural constraints to an addition process that, unlike string-based semantics, does not include ordering within the concept of addition. To add several stacks of pennies, a child may pick them all up with both hands, and then place a single stack down on the table. The number of stacks being combined is structurally irrelevant; the presumed sequence of picking up coins one pile at a time is temporally irrelevant.

Further, the particular pennies in any of the stacks added together are indistinguishable. Within a stack there is no particular ordering, no penny that is distinguished as the first or the last in the stack. (Well, penny stacks do require a physical top and bottom coin, in that way the analogy is weak. We could equally well use a bag or a handful of coins as the unordered physical structure, what we call an ensemble of coins.) We will appeal to the handful analogy, we are consider a group of pennies, and other groups of pennies in other hands, and they all become a single group of pennies by being concurrently dropped into a larger bag.

The central principle is that, from the perspective of the addition operation, no particular penny, or group of pennies, is special. There should be nothing within the specification of what it means to add that says that one particular added unit is more central than any other added unit. The concept of commutativity has this idea at its heart, but since commutativity conventionally must be asserted, it starts at the wrong place, by assuming a difference (when there is none) and then correcting this misconception by making a commutative rule.

Pennies-on-a-table provides a concrete example of a unit arithmetic, for which magnitude is expressed in a unary rather than a binary or decimal base. The example illustrates that, conceptually, addition can be formulated outside of the structural rules of modern algebra, that indeed the rules of algebra may do disservice to the simplicity of the idea of adding. We will show that addition can be formulated without exchanging the Principle of Addition for the conveniences of symbolic abstraction. And we will show this more natural formulation to be both rigorous and efficient.

Both Inside and Outside

The fundamental idea in boundary mathematics is to express mathematical concepts using representations that have both an inside and an outside. A circle () is

the prototypical planar boundary form. String tokens have only an outside, so that concatenation of string tokens is defined as adjacency on the outside. Boundary symbols, such as circles, can be composed on the outside, () (), and on the inside, (()). One consequence of "two sided" symbols is that they can be interpreted both as objects (when composed on the outside) and as operators (when composed on the inside).

It is important to accept that a boundary symbol is atomic, and not a relation between inside and outside. As an atomic structure, a boundary symbol is concurrently object and operator, the specific interpretation is lifted from the representation itself, becoming a semantic choice when a form is read. That is, in boundary mathematics, there is no explicit differentiation between object (member of a domain or codomain) and operator (function assigning correspondence between domain and codomain members).

When a boundary form is read as an object, transformation can be defined by substitution of equivalent objects. Objects can be generalized to patterns by including variables to represent arbitrary forms. For example, the idempotency pattern gives permission to delete all replicate forms sharing a space. Below, this pattern is illustrated for atomic objects and for patterns:

() () = ()

A A = A

Depth-value notation, in Section %%%, use an enclosing boundary extensively as an operator, one that multiplies its contents by the base value of the numerical system.

Unit arithmetic provides a running example of the more general idea of the spatial formalism of boundary arithmetic. Boundary arithmetic is a calculus of partial orderings, using the frames that contain featureless space as a syntax for nested and juxtaposed abstract spaces. For pennies-on-a-table, the perimeter of a coin serves as a boundary, or frame, that differentiates what is outside the coin from what is "inside". Coins stacked together are inside the stack's boundary, while non-stacked coins are outside. Both conceptually and physically, the stack boundary defines a third dimension of representation, independent of the two spatial dimensions of the table top.

Representations that possess both an outside space and an inside space participate in a substantively different type of syntax than those (such as string tokens) that have only an outside. We are familiar with this idea from the common notation for a function, f(x,y), which represents an operation when read conventionally from the outside, and which represents the arguments of the operation when read inside the function's argument delimiters. When an argument is itself a function, such as g(a,b), the space inside the function delineators

is nested further by an application of function composition, f(g(a,b),y). In functional notation, the space of arguments permits expressive options (such as the number and order of arguments) not available to tokens having no interior.

In boundary mathematics, the representation of constants and variables incorporates both an outside and an inside perspective. Read from outside, the example of a penny is unitary. Read as part of a stack, that penny participates in a grouping. Thus, each atomic unit has a dual role, as an object in relation to other units on the outside, and as an object in relation to units inside a common shared stack. We explore several substantive differences between string-based and boundary representations in the sequel, the most fundamental of which is that the spatial zero is implicit as empty space. The explicit representation of nothing as a token is both unnecessary and undesirable.

Increasing the dimensionality of a representation creates a one-to-many map between spatial and string syntax. Further structural constraints can be appended to spatial forms to construct an isomorphism to string forms; such elaborations are equivalent to converting the spatial formalism to a string formalism. Herein, we pursue the more adventurous path of expressing integer arithmetic in a purely spatial syntax, the advantage (or cost, depending upon one's commitment to string processing) is that much of the structural foundation of modern algebra becomes irrelevant.

Unit ensemble arithmetic is our running example of a boundary mathematics. But to illustrate the generality of the techniques, we now present an application to elementary algebra.

SPATIAL REPRESENTATION of ELEMENTARY ALGEBRA

Our understanding of a concept is tightly connected to the way we represent that concept. Traditionally, mathematics is presented textually. As a consequence novice errors, in elementary algebra for example, are due as much to misunderstandings of the nature of tokens as they are to miscomprehensions of the mathematical ideas represented by the tokens. We interact with spatial representations through natural behavior. When the environment enforces the transformational invariants of algebra, the spatial representation affords experiential learning.

In general, how we represent numbers is a matter of convenience. For learning mathematics (and for doing mathematics) it is often more convenient to call upon visual interaction and natural behavior than it is to conduct symbolic substitutions devoid of meaning. Spatial algebra uses the three dimensions of natural space to express algebraic concepts. A higher dimension of representation greatly simplifies the visualization and the application of algebraic axioms. Algebraic transformation and the process of proof are achieved through direct manipulation of the three-dimensional representation of the algebra problem. Spatial algebra addresses common errors made by novice algebra students by permitting experiential interaction with abstract representations.

===
The visual programming community has developed taxonomies of visual approaches [13]. The experiential approach to mathematical formalism presented in this paper is sufficiently unique not to fit into existing taxonomies of visual languages. The approach of mapping formal operations onto the topological structure of space itself is not diagrammatic, iconic, or form-based. Most fundamentally, experiential mathematics imparts semantics onto the void (empty space). Actively using the void is both simple and conceptually treacherous [3].
===

A Spatial Algebra

The components of space which can be used for the representation of mathematical concepts include:

- empty space (the void),
- partitions between spaces (boundaries, objects),
- labeled objects that share a space, and
- labeled objects that share a boundary (touch one another).

This is sufficient structure for the expression of elementary algebra. One possible map from algebraic tokens to algebraic spaces is:

```
      Constants:
                        { 1,2,3,...}  -->   { labeled-blocks }
      Variables:
                        { x,y,z,...}  -->   { labeled-blocks }
      Operators:
                        { + }         -->   { sharing-space }
                        { * }         -->   { sharing boundaries }
      Relations:
                        { = }         -->   { partitions of space }
```

Examples of a spatial representation of the above map follow.

Constant as labeled block:

```
                                  3
```

Variable as labeled block:

```
                                  x
```

Space sharing as addition:

```
                    3 + 2 = 5
```

Touching as multiplication:

```
                          3 * 2 = 6
```

A simple algebraic term:

```
                               2x + 3
```

The gravitational orientation of the typography (top to bottom of page) in the above examples is not an aspect of spatial algebra, although gravitational metaphors are useful for the representation of sequential concepts such as non-

commutativity. As well, the sequencing implied by stacked blocks is an artifact of typography; stacks only represent groups of objects touching in space.

Group Structure of Spatial Forms

Elementary algebra is taught in high schools as an application of a very general theory called the theory of groups. A group is simply a set of objects and an operation that converts some objects into other objects. Groups can have particular properties or characteristics. The Rules of Algebra as taught in school are the properties of addition, multiplication, and equality as defined by group theory.

Generally, spatial representation can be mapped onto group theory. A commutative group is a mathematical structure consisting of a set and an operator on elements of that set, with the following properties:

- The set is closed under the operation.
- The operation is associative and commutative.
- There is an identity element.
- Every element has an inverse.

The integer addition and multiplication operators belong to the commutative group.

Commutativity

Spatial representation permits the implicit embedding of commutativity in space. The commutativity of addition is represented by the absence of linear ordering of blocks in space (visualize the blocks in this example as floating in space rather than in a particular linear order):

$$x + y = y + x$$

We intuitively recognize objects contained in a three-dimensional space as ordered solely by our personal perspective. In contrast, typographical objects are necessarily ordered in sequence by the one-dimensional nature of text and by the two-dimensional nature of the page.

Commutativity of multiplication can be seen as the absence of ordering in touching blocks:

$$x * y = y * x$$

Again, in space there is no preferential ordering to touching objects:

Associativity

Associativity of addition is the absence of an explicit grouping concept in space:

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

The apparent visual grouping expressed by differences in metric distance between blocks can be assigned a semantics of associativity (for example, add closest objects first), or it can be ignored, permitting the operation assigned to space to address multiple arguments in parallel. From an intuitive perspective, operations embedded in space apply to any number of objects in that space. Whatever grouping we use is a matter a choice and convenience. Parallel computers provide techniques for addressing all objects at the same time.

Associativity of multiplication is the absence of an explicit grouping concept in piles:

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

The apparent visual ordering of piles can be overcome by assuming that all objects in a pile touch one another directly. Rather than displaying stacked objects, we might present objects in piles as completely interpenetrating. Every object in this non-physical representation is in contact with every other object, forming a Cartesian product of touching objects.

Distribution

Precedence operations associated with the distributive rule are the most common algebraic error for first year students [12] [4]. The representation of distribution in spatial algebra is particularly compelling. Generally, the distributive law permits combining blocks with identical labels into a single block with that label. Conversely (read right to left), distribution permits splitting a single block that touches separate piles into separate but identical blocks touching each pile:

$ax + bx = (a + b)x$

Blocks with identical labels are both singular and arbitrarily subdividable in space. This ability to arbitrarily divide and combine blocks with a common name is the same as the ability to arbitrarily create duplicate labels in a textual representation. Changing the size and the number of occurrences of a labeled block is easy in a virtual environment.

Any potential ambiguity between distributive idempotency and the use of space as the addition operator is avoided by the effect of context on interpretation. Idempotency requires the context of touching blocks (multiplication). Addition requires the context of non-touching piles.

Identities

Zero is the identity element for addition. The identity in the spatial metaphor is the void; identities are equivalent to empty space.

The additive identity:

$$x + 0 = x$$

That is, zero disappears in space:

The multiplicative identity:

$$1 * x = x$$

The One block disappears only in the context of an existing pile. A zero in a pile makes the entire pile disappear:

$$0 * x = 0$$

Additive Inverse

The inverse of a positive number is a negative number. Negative numbers are the most difficult aspect of arithmetic for elementary students. One way to directly

represent inversion is to create an inverter block. Another way is to create an inversion space; for example using "under-the-table" for inverses. Inverses can be represented in many ways: as inverters, as colors, as orientations, as different spaces, as binary switches, as dividing planes, as inside-out objects.

In this version of spatial algebra, piles are inverted by the inclusion of a special inverter block:

Since a negative number can be seen as being multiplied by -1, the inverter block is expressed as touching (multiplying) the pile which is inverted:

$$-x = (-1) * x$$

The inverter block expresses subtraction as the addition of inverses,

x - x is written as x + (-x)

The additive inverse:

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

Calculus of Signs

The use of the inverter block for negative numbers introduces a calculus of signs into the algebra of integers. A sign calculus requires the explicit introduction of the positive block:

The positive block is the inverse of the inverter block. It introduces the concept of polarity and the act of cancellation. Numbers without signs are usually assumed to be positive. Making signs explicit removes this assumption.

The following rules of sign calculus assume each sign has a unit value associated with it.

Additive cancellation in space:

Cardinality in space:

Multiplicative cancellation in piles:

Multiplicative dominance in piles:

The following example illustrates an inverter sign distributed across all objects in a space:

$$(-x) - y = -(x + y)$$

Multiplicative Inverse

Finally, division is the multiplicative inverse. Again, there are many possible ways to represent an inverse in a spatial representation. Since the traditional notation for fractions is primarily two-dimensional, it already has many spatial aspects. The division line that separates numerator from denominator could be carried over to the spatial representation as a plane dividing a pile into two parts. Here however, the multiplicative inverse is represented by inverse shading of the block label:

$$1/x$$

The multiplicative inverse:

$$x * 1/x = 1$$

One weakness with the choice to represent a reciprocal as differently shaded labels is that composition of reciprocals -- for example 1/(1/x) -- is not visually defined. Choice of representation necessarily effects pedagogy. It is

an empirical question as to which representations facilitate learning algebraic concepts efficiently.

Fractions are the second most difficult area for students of arithmetic. A typical problem using fractions requires the application of the distributive rule:

```
a/b + c/d = (ad + bc)/bd
```

Factoring

Factoring polynomial expressions is equivalent to multiple applications of distribution. For instance:

```
x^2 + 4x + 3 = (x + 1)*(x + 3)
```

One advantage of the spatial representation on the right-hand-side of this equation is that both the factored and the polynomial forms are visible concurrently. Looking from the side, we see two completely touching spaces which represent the factored form:

```
(x + 1) * (x + 3)
```

Looking down from the top, we see four piles which represent the polynomial form:

```
x^2 + 1*x + 3*x + 1*3
```

Here, the factored form is converted to the polynomial by slicing each addition space through the middle.

Caveats

The representational details of the spatial algebra presented here are, like any choice of syntax, somewhat arbitrary. We lists many options, for example, for the representation of inverses. This representational freedom can be constrained by empirical studies intended to determine which particular representations are effective for task performance. There is no reason to believe that effective representations are generic. More probably, different individuals will prefer and understand different representations in the context of different tasks. Still, the research to determine which representations are

effective has yet to be conducted. In fact, demonstrating that spatial algebra actually improves performance in high school algebra remains as future research.

The weakest aspect of the proposed spatial algebra is the representation of three or more multiplied objects, x*y*z for example. This form can be represented by either completely interpenetrating blocks or by "blocks" with complex shapes that twist around to touch all other blocks. This problem gets particularly difficult for multiplying several factored expressions, for example: (x + 1)*(x + 2)*(x + 3)

In general, the cubic blocks presented above are misleading, since they imply a Cartesian coordinate system. In fact, the spatial representation proposed here has no associated metric (or rather, the metric is irrelevant to the mathematical formalism). The treatment of space might be improved by explicitly including a representation for the table which blocks can be imagined to rest upon.

Spatial representation provides a map to a wide range of new visual languages. The examples above are expressed in a language of labeled blocks. The spatial rules, however, map just as easily onto people in a room, toys in a box, salmon in streams, and bricks in a wall.

Principles of Spatial Mathematics

We use the term boundary mathematics to describe the collection of rules and tools used to generate representations of spatial algebra. Boundary mathematics is general in that its principles can be applied to many mathematical domains. Boundary mathematics is quite unique, since it incorporates both the participant and the void into its formal structure.

General Principles

-- Mathematics is the experience of abstraction.

-- Experience is not a recording. Representation is not reality.

-- The void cannot be represented.

-- Space requires participation. To participate is to partition space, to construct a boundary.

-- Boundaries both separate and connect.

-- Boundaries identify an intentional construction.

-- Representation and meaning are different sides of the same boundary.

-- Our body is our interface.

Mathematical Principles

-- Operators, invariants, and identities can be embedded in space.

-- Multiplicity is generated by observation.

-- Commutativity is embedded in space, ordering is embedded in time. Spatial entities are asynchronous parallel processes.

-- Associativity is the choice of the participant. Spatial entities are autonomous.

-- Entities are both singular and plural in form, depending upon the construction of the participant. Entities with the same name are the same entity.

-- That which is common to every entity in a space is common to the space itself, forming the ground of the space.

-- Touching spaces are in pervasive contact (Cartesian product).

-- Crossing a boundary inverts a space. Inversion unites partitioned spaces.

-- Normalized spaces are those equivalent to the void. They can support arbitrary grounds.

==========

[2] Bricken, M. (1991) Virtual reality learning environments: potentials and challenges Computer Graphics Magazine, ACM, 7/91.

[3] Bricken, W. (1986) A simple space Proceedings of the Sign and Space Conference, University of California at Santa Cruz. Also as HITL Technical Report R-86-3 , University of Washington, 1989.

[5] Bricken, W. (1989) An introduction to boundary logic with the Losp deductive engine Future Computing Systems 2-4.

[6] Bricken, W. (1992) Virtual reality: directions of growth Proceedings, Imagina'92, Centre National de la Cinematographie, Monte-Carlo.

[10] Kauffman, L. (1980) Form dynamics Journal of Social and Biological Structures 3, 171-206.

[12] National Assessment of Educational Progress (1981) Results from the second mathematics assessment National Council of Teachers of Mathematics, Reston, Va.

[13] Shu, N. C. (1988) Visual programming Van Nostrand Reinhold, New York.

[16] Spencer-Brown, G. (1972) Laws of form Julian Press, New York.

[18] Winn, W. & Bricken, W. (1992) Designing virtual worlds for use in mathematics education Proceedings of AERA, 1992.

UNIT AND BOUNDARY ARITHMETICS -- SECTION IIC October 1, 2008 10:55 PM
William Bricken
January 2007

Boundary Mathematics
- Unit Arithmetic
- Unit Arithmetic in Antiquity
  - Sumerian Cuniform
  - Egyptian Hieroglyphics
  - Roman Numerals
    - Pictographic Roman Numerals
    - Readability

UNIT ENSEMBLES
- Dot Pictures
- Foundations
- Void Foundations
- Counting vs One-to-one Correspondence
- Calculating without Counting
- Boundary Delineation
- Names and Variables

Boundary Mathematics

Boundary mathematics is the study of formal systems that utilize diagrammatic, planar, spatial, and physically manipulable representations, in contrast to the token-string representations of conventional formalisms. For example, the Euclidean School of Geometry in ancient Greece [ref] used concrete diagrams to define geometric axioms.

==finish

For example, boundary arithmetic relies upon syntactic constructs that occupy the two dimensions of a page; representations are, so to speak, positionally free. In particular, boundary arithmetic is independent of the concepts of commutativity, associativity, and arity. Imputing these positional, temporal, and numerical syntactic constraints is a violation of the intended semantics of boundary arithmetic; specifically, boundary mathematics treats the ordering, grouping, and arity of arithmetic functions as semantically irrelevant structural detail. An unintended but unavoidable consequence is that boundary mathematics rests firmly outside of the conceptual structures provided by modern algebra, group theory and set theory. Although not emphasized here, it also stands outside of conventional sentential logic and Predicate Calculus.

===delete?
Algebra

The representations and transformations of boundary arithmetic are readily converted in a boundary algebra with variables. Substitution and replacement are core operations within an algebra, so that boundary arithmetic prepares for the pedagogical transition from arithmetic to algebra. Figure 11 show one possible extension of boundary arithmetic to algebra, by applying boundary notation to Peano's axioms.
===

Unit Arithmetic

===
CONSTRUCTOR

• isan ensemble
if A and B are ensembles then so is A B
No others
===

suggest UA is easier than positional notation

Consider unit-arithmetic:

to count: establish a one-to-one correspondence between units and cardinal numbers
to unit-add: place collections of units together
to unit-multiply: replace each unit of one form with the other form

These operations are commonly taught using Cuisenaire rods and base-10 arithmetic blocks. Unit arithmetic is base-1. Place-notation does not make sense in a base-1 system, since each new unit requires a new position. The absence of place-notation leaves units free to be placed anywhere on the surface of representation, wether it be a page of a table. This of course means that adding groups of units does not require any effort in keeping track of which units represent which magnitudes. The difference between adding by pushing piles together and adding by keeping track of magnitudes and aligning and carrying it purely notational, and has nothing to do with what numbers mean.

By grouping blocks into collections of the same size, unit-arithmetic can be converted into an analog of Roman numerals. By restricting the size of collections to one magnitude, a base system is added to unit-arithmetic. However, these extensions effect the reading of unit-numbers, they do not change the operations of addition and multiplication.

Both unit-addition and unit-multiplication follow the Additive Principle: the combination of the parts represents the sum, without further computation. The price, of course, is that the answer must be retrieved from the combined pile of blocks by (re)reading the piled blocks in one-to-one correspondence to the whole numbers.

Unit-arithmetic does not translate well into the group theoretic structural rules of modern algebra. The additive identity in unit arithmetic is "doing nothing", adding no blocks to a specified pile. Commutativity is difficult to discern when piles are pushed together without selectivity. Associativity too is difficult to support when multiple piles are pushed together simultaneously.

And usually, the additive inverse is converted from a numerical object to a process of "taking away" blocks.

Since unit-multiplication is defined in terms of making a substitution, the order of substitution is structurally relevant, making the commutative and associative rules of multiplication also relevant. The multiplicative identity is substitution of units and wholes.

The additive and multiplicative inverses define new objects, expanding the domain of application from whole numbers to integers to rationals. These new objects can be added to unit arithmetic, generating unit-integer and unit-rational systems.

Figure 2 shows the mapping of unit arithmetic to the concepts of group theory. We are observing that unit arithmetic do not map onto the group theoretic concepts of place-value arithmetic, especially for addition. The issue is deeper than syntax, unit arithmetic does not include the group theoretic structures of symbolic addition. The weakness of the map may manifest at the level of a student's understanding and constructive modeling, or at the level of technical mapping of symbol systems and their semantics. The divide between concrete and symbolic manipulation of numbers is not superficial, the two systems are simply incommenserable, an observation generally ignored by the mathematics curriculum.

In math education texts, for example, commutativity of unit-addition is achieved by fiat. "We may associate 3+5 with putting a set of 3 members in a dish, and then putting a set of 5 members in a dish to form the union of the sets. We associate 5+3 with putting the 5 set in a dish and then putting in the 3 set." [ref johnson] Unit-addition, however, does not require or incorporate an external dish or a temporal ordering or actions. When a 3 set and a 5 set share the same table, children will add them by pushing the piles together concurrently. One set simply does not have temporal precedence over another. More radically, it is possible to add the two sets merely by shifting one's attention, from the individual sets to the contents of the table. All that the additive principle requires is a shift of the focus of attention.

===
integrate with above
===

An example of a boundary arithmetic is tally or stroke arithmetic, within which identical marks are recorded in one-to-one correspondence to objects. Figure %%% shows the representation of the first few whole numbers in three tally systems, Pebbles, Tallies, Stars, and Dots. Regardless of the type of unit, these systems all share the common characteristic that the magnitude of a number is explicitly represented by a one-to-one correspondence with indistinguishable unit "objects". The multitude of unit markings explicitly represents the cardinality of an intended number.

Here we call the arithmetic of identical marks, unit ensemble arithmetic, unit arithmetic for short. We represent units by centered dots, •. Unit ensemble arithmetic provides a continuing example in the sequel, however the formal principles of the boundary mathematics underlying unit arithmetic apply broadly to numerics, to logic, to algebra, and to more exotic systems such as knot theory and fractals.

	Pebbles	Tallies	Stars	Dots
1		/		•
2		//		••
3		///		•••
4		////		••••
...				

Figure %%%: Unit Arithmetic Numbers

Unit arithmetic possesses specific structural characteristics that identify it as a spatial formalism. These spatial properties lift the representation of arithmetic from the strict ordering of token-strings into the space of diagrams, drawings, and other iconic representational forms. The units, or marks, within a unit arithmetic are recorded in an open planar space that is free of an underlying metric. Magnitude is directly visible as the accumulation of a quantity of indistinguishable marks.

Addition in unit arithmetic is characterized by the Additive Principle: a sum is represented explicitly by the accumulation of its components, or parts. Figure %%%. Symbolic arithmetic, in contrast, represents a sum as an abstract encoding derived from transforming the abstract encodings of the components of the sum.

==========
Fig. %%%:
==========

Multiplication of two collections of marks in unit arithmetic is by substituting a replicate of one entire collection for every unit in another collection. Figure %%% show how multiplication can be easily visualized. Although marks can be subtracted by "crossing them out", unit arithmetic does not have an explicit zero token. Zero marks is not represented as 0 or as any other token. It is instead the absence of marks, the presence of unmarked space. The semantic use of non-representation is a signature characteristic of a spatial, or boundary, mathematics.

==========
Fig. %%%:
==========

Unit Arithmetic in Antiquity

Archeological evidence [ref Schmitt-Besserant] suggests that unit arithmetic was in use in pre-Sumerian cultures, circa 5000 BCE. By 4000 BCE, units were combined into groups of equal size, most probably to facilitate counting. Base-10 grouping is dominant in ancient numerical systems, as is positional notation. In positional notation, tokens for grouped numbers are arranged by order of magnitude in a line. A primary (but not particularly ancient) example is Roman Numerals. Three thousand years prior to the Romans, Sumerian cuniform also included grouping tokens.

Sumerian Cuniform

Egyptian Hieroglyphics

Roman Numerals

The Romans iused names for collections of certain numerical values. What we call "five" was spatially converted to the shape "V". For example,

5 == V == /////

The Roman numerical names (translated into their modern reading) include:

I == 1
V == 5
X == 10
L == 50
C == 100

```
D  ==   500
M  ==  1000
```

The particular numeral assignments have deep roots in the pre-history of the evolution of mathematics through counting (called unit form above). The techniques of numerical arithmetic evolved in co-dependence with our built-in numerical computing tool, the hands. The symbolic representation of these numerical processes arose in co-dependence to other physical tools, such as impressions in clay, knots in string, and the abacus.

Pictographic Roman Numerals

Roman numerals are not Roman letters, although capital letters do serve as a convenient typographical form of numerals. C, which represents the numeral 100,

ccI>>

cccI>>>

The Roman numerals for large numbers are boundary forms. The letter sequences that depict them are iconic forms for spatial structures, and not letters of the Roman alphabet (">" is reversed "C"):

100	C	
500	I>	
1,000	CI>	(|)
5,000	I>>	|))
10,000	CCI>>	((|))
50,000	I>>>	|)))
100,000	CCCI>>>	(((|)))

Early versions of these ideas were present in Roman numerals, for example, where 729 would be represented as DCCXXIX (D=500, C=100, X=10, and I=1). Although Roman numerals use grouping by tens and the interpretation of a numeral depends to some extent on the placement of the symbols,20 they do not at all constitute a place-value system. Also, the system of Roman numerals is ad hoc, in the sense that each new grouping requires a new symbol, so it is strictly limited in extent.

Readability

The problem of readability of large numbers is addressed by having names for large groups, such as M for 1000. Like unit arithmetic, Roman Numerals can be added by collecting representations together in a shared space:

1096 + 387 == MLXXXXVI + CCCLXXXVII = M CCC LL XXXXXXX VV III

The result of a sum is significantly easier to read than the corresponding sum in unit ensemble notation, which would consist of 1483 dots. Figure %%%.

==========

Roman Numeral and Unit Ensemble Arithmetic Addition

(spatial joining w/wout groping) 1483 DOTS

Figure %%%

==========

Specific "Roman Numeral facts" improve the ease of reading even more. For example:

5+5 = 10 == VV = X
10+10+10+10+10 = 50 == XXXXX = L
50+50 = 100 == LL = C

1483 == M CCC LL XXXXXXX VV III = M CCCC L XXX III

Although Roman numerals use grouping by tens and the interpretation of a numeral depends to some extent on the placement of the symbols,20 they do not at all constitute a place-value system. Also, the system of Roman numerals is ad hoc, in the sense that each new grouping requires a new symbol, so it is strictly limited in extent.

===

We next develop a formalism for spatial units, without as yet calling upon an interpretation as an arithmetic of integers. Spatially arranged units constitute a boundary mathematics. Units are both located outside one another, and located inside a container that defines them as a Whole, an ensemble of identical parts. We adopt the whole/part relational structure from mereology. We then describe the structure of equivalence relations upon spatial forms, in particular defining substitution of forms within a given spatial environment as a primary transformation mechanism.

===

UNIT ENSEMBLES

A unit is a single mark, stroke, tally, pebble, or other singular distinction. We represent the atomic unit as a centered dot, • . Unit marks may be replicated, providing a supply of indistinguishable replicas. We will enforce the indistinguishability of unit replicas. The only difference between replicated units is the space that each occupies. Replicas have separate existence, but cannot be indexed, labeled, or uniquely identified. Of course, the constraint of singular representation applies only within one computational or representational context. We can freely rewrite a configuration such as that in Figure %%% on another page. More commonly, spaces separated by an equal sign can each reference the same object.

Units can occupy space in two distinct ways, by SHARING SPACE and by being PARTITIONED into different spaces (that is, by not sharing the same space). Figure %%% shows three separate spaces with several units sharing each space.

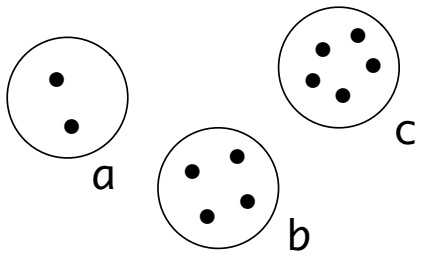


Figure %%%: Distinguishable Spaces and Indistinguishable Units

Units that share the same space constitute an ensemble. A single space can contain only one ensemble of units, specifically the units that share that space. Units are components, or parts, of an ensemble, as well as discrete entities. An ensemble can consist of a single unit, but it cannot consist of no units. When units are absent, ensembles do not exist. Thus there is only one type of form, the ensemble, that is co-defined with the space it occupies. We will consider only transformations on ensembles.

Note that ensembles differ significantly from sets;

-- there is no "empty" ensemble
-- the parts (members) of an ensemble are all identical
-- the parts are not separately identifiable objects
-- there is no distinction between a single unit
and an ensemble consisting of a single unit part, and
-- no unit participates in more than one ensemble.

Dot Pictures

For typographical convenience, we will place units that share the same space together in a continuous line, for example, ●●●●●●●. Stringing units together is an informal notation and does not imply a string syntax.

==========
fig circles and dots
==========

It is important to appreciate that units in an ensemble share absolutely no relative, metric, or positional relationships. Since units in an ensemble cannot be differentiated, they cannot participate in semantic relations. Thus, the concept of the intersection of ensembles is not defined, for if intersection were possible, then at least one unit could participate in two ensembles. This unit would be "in two places at the same time", and thus would be distinguished from other units.

An alternative syntactic method of conceptualizing units in ensemble is that we are not free to construct replicate names, a unit cannot have pointers to it. Units are not referents of variables; they represent themselves. This perspective treats the notion of uniqueness of reference seriously: each separate representation of a unit is a different unit. A unit that exists refers only to itself. Thus, units have a significantly different semantics, or meaning, than do conventional tokens. Specifically, the unit dot indicates a unit that connects to nothing but itself. It is its own meaning.

Should we wish, the complete non-interaction of units can be attributed to the space inside an ensemble boundary. That featureless space has no underlying substructure to support interaction between contents. This idea is not unusual, it is the semantics of sets. The defining characteristic of a set is its membership.

The calculus of unit arithmetic does not include labels for or variables over units. Thus, for example, it is syntactically not possible to write a commutative rule for units, just as the commutative rule applied to unit integers loses its interpretability:

1 + 1 = 1 + 1

A deeper issue addressed in the sequel is that we no longer have the convenience of predicate calculus as a base language for formalization. In particular, the concept of a unit ensemble requires non-standard functions and relations.

Foundations

Replicate units are intended to be indistinguishable, in order to reduce the idea of counting to a foundation of one-to-one correspondence between marks and objects. The semantic equality between an abstract cardinal Number and the quantity of its referent is then defined only by an operation of matching identical units to things one-to-one. This approach differs from that of set theory, in that sets require unique symbolic forms to correspond to unique objects. Number is a property of an infinite class of sets that each has exactly the same "number" of members.

With unit ensembles, quantity is achieved by replication rather than by unique identification. The replicate function shares much with the successor function of conventional axioms of numerics. The successor function constructs and identifies a unique individual that is "next". This new number is strictly connected with the "prior" number from which it sprang. The replicate function maintain no such allegiance. There is nothing new constructed, replicate is but a mirror conveniently placed so that we might perceive more than one.

Void Foundations

>< >< not

• = •

The significant difference is that the addition and multiplication tables for binary form requires entries for 0. Thus there are four possibilities for a particular sum of two binary digits:

```
0+0 = 0
0+1 = 1
1+0 = 1
1+1 = 0     carry 1
```

This translates to an XOR gate in circuitry, a single logical function that, of the two-input logical gates, requires the most complex wiring of transistors.

In boundary forms, the literal absence of 0 reduces the computational transformations to one. In the case of "0+0", absence does not support the concept of multiplicity. That is, one cannot tell if nothing has been added to nothing, or if it hasn't. Similarly, in the cases of "0+1" and "1+0", one cannot determine if the unit that is present has or has not been added together with nothing. Thus the only computational case is that of "1+1", that is, unit merge. In circuitry, unit merge can be determined by a simple AND gate. &&?&&

The boundary situation for addition is similar to binary multiplication, for which absence of two forms determines the result:

```
0x0 = 0
0x1 = 0
1x0 = 0
1x1 = 1
```

Boundary multiplication, however, does not involve "times tables", multiplication is achieved by simple substitution for units. For forms with more than one unit, multiplication also requires replication. In circuitry, this is routing of signals (for substitution), and splitting of signals (for replication).

Since unit substitution occurs at the depth of the particular unit, multiplication via addition of exponents occurs automatically. Multiplication of magnitude is always 1xN, the single case of look-up which is significant in binary multiplication.

===
leibniz-clarke
===

An accurate assertion of the idea that every unit is indistinguishable would be the algebraic rule

$$\bullet\bullet = \bullet$$

which reduces all unit representations to their singular identity. Conventionally, this is an idempotency rule, and leads to a spatial Boolean algebra by collapsing the idea of separable replica.

Here we wish to construct multiplicity via replication of indistinguishables, which can be achieved formally by denying that the space containing units is idempotent:

$$\bullet\bullet \neq \bullet$$

Instead, uniqueness of reference is supported by the concepts of ensemble and of disjoint partitions.

We thus permit replications to continue in existence, while still requiring that they are in all other ways indistinguishable. We will, however, need a mechanism to distinguish ensembles of units; this can be achieved spatially by providing a capability to maintain different spaces. In string representations, a space between letters defines a word. Although letters are the atomic unit of

syntax, words are the atomic unit of semantics. It is the space between words that distinguishes structure from meaning. Similarly, we will distinguish units within an ensemble by proximity. In a spatial notation, proximity can be spread over the plane of the paper. To be clearer still, we place a spatial boundary around each ensemble that is assigned a distinguishable meaning, or reading, as is illustrated in Figure %%%.

==========
Fig. %%%:
==========

Counting vs One-to-one Correspondence

The constraint that units cannot be distinguished from one another can be phrased in another, somewhat surprising, way: we are not able to count units. Counting requires assigning each unit a unique identifier, that is, it requires making them distinguishable. The calculus we are developing is sufficiently simple to operate prior to the idea of counting. The technique that replaces counting is one-to-one correspondence. Counting is, in fact, placing units in one-to-one correspondence with a sequence of unit increments. What we will not be using, and for spatial arithmetic do not need, is the sequence of increments otherwise known as the integers.

The proposition that we do not need the integers for arithmetic sounds quite strange. However, when numbers are represented by unit ensembles, we have the ensemble itself to stand in place of the unique integer it corresponds to. This approach is quite visual, an ensemble illustrates the Number of its unit parts. An ensemble is also abstract, in that a particular ensemble corresponds to a Number; it is not, for example, a member of a collection of ensembles.

Cardinality is the only property that distinguishes ensembles. Within the algebraic frame of mind,

$$\bullet \; \bullet \neq \bullet$$

is the only relational rule.

The Number of unit replicas within an ensemble can be determined only by applying a counting function that works through one-to-one correspondence. Number is an external characteristic of ensembles.

A natural concern would then be: how can we know the number of units in an ensemble without counting them? And of course, we cannot. The idea though is that we can separate the operations of an arithmetic calculation from reading

the result, and we need to count only once, when we have arrived at the result of potentially many operations.

Calculating without Counting

Unit ensemble arithmetic provides all the machinery needed to do conventional arithmetic operations without having to stop to count. After the operations have been completed, it is necessary, only once, to count what remains as the answer to the computation. The innovation is primarily one of separating the idea of arithmetic calculation into two independent components, transformation and evaluation. The spatial representation of unit ensembles permits this separation of tasks. We might say that conventional arithmetic is easy to read (to count) but not so easy to calculate with. Unit ensembles are easy to calculate with, and not so easy to read.

===
Money as an example? don't count pennies in quarters
===

The difficulty in reading unit assembles is solely in tallying up the individual units in an ensemble that represents the result of a computation. The difficulty in computing with conventional Arabic/Hindu numbers is that we are maintaining the capability of easy reading throughout the computational process. It is quite clear that both representations of the integers address the same semantics, that of the transformation of Numbers under the operations of arithmetic. Any Arabic/Hindu numeral can be decomposed into a sum of units, and thus brought into correspondence to unit ensembles. Similarly, any unit ensemble can be encoded via counting as a conventional place-valued numeral. The specific issue is not that the abstract systems are different, it is that the concrete representations are different. And, somewhat contrary to the conventional mathematical position that syntax does not interact with semantics, syntax does interact with the algorithms that achieve semantic transformation. By strictly separating reading from operation, we can engage in simple spatial algorithms during the transformation stage, and then simple one-to-one counting during the reading phase. See Fig %%%.

==========
Fig. %%%:
==========

Another concern may be that computing with unit ensembles is indeed difficult, since we must deal with a lot of them, particularly for large numbers. In Section %%% we will provide a base-10 spatial notation for unit ensembles that maintains the simplicity of the transformational rules while gaining the representational efficiency of common place-value notation. We preserve the

ease of spatial transformation of unit ensembles through a spatial depth-value notation that has all the advantages that conventional place-value numerals confer upon reading, without the algorithmic overhead that place-value notation places upon calculation. By comparing the two approaches, we can possibly learn more about the nature of number.

Table %%% presents a comparison of various notations across readability, computability, and convenience.

Boundary Delineation

Boundaries can also be expressed in string format by parentheses, braces, and brackets. [quote Knuth] For typographical convenience, we will represent ensembles as bounded dot pictures, and more than one ensemble as bounded dot pictures with a separating bar, | , between ensembles to indicate separate and disjoint spaces. Bars that separate ensembles are partition delimiters. The bar is an informal notation that permits a more compact representation of discrete ensembles. Thus

(●●●|●●●●)

represents two ensembles in two different spaces. For convenience, we omit the containing boundary around singular wholes, letting the implicit boundary of white space delineate the ensemble.

Different ensembles must necessarily occupy different spaces, since all unit parts within a particular space participate in the same ensemble. Ensembles are disjoint in space, so that the idea of a part "on the left of a bar" and a part "on the right of a bar" has no meaning. More appropriately in spatial syntax, Figure %%% illustrates that there are no positional metrics associated with the representation of partitioned Wholes:

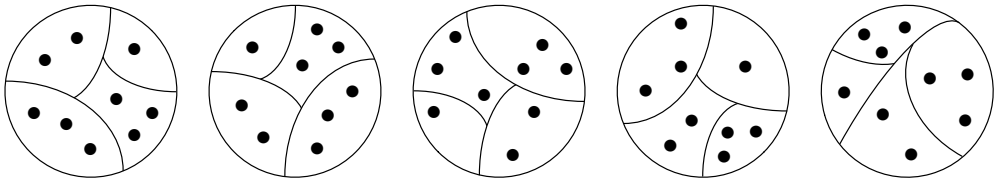


Figure %%%: Several Indistinguishable Partitioned Wholes

Each of the partitioned Wholes in Figure %%% is identical. The varieties pictured in Fig %%% illustrate, respectively, that rotation of a Whole,

placement of units within partitions, placement of partitions within Wholes, and shape of partition boundaries have no intended semantics within the spatial syntax. The variety of appearance over representations of a given partition in analogous to the variety of appearances afforded arbitrarily selected variable tokens within a string representation.

Typographically, each of the above circle pictures is represented by one dot picture:

(●●●|●●●|●●|●)

The above partition of ●●●●●●●●● is a singular structure consisting of a collection of wholes within the ensemble delimiter (represented by parentheses in the example). A partition is not a set, and also not a multiset, for several reasons:

-- an empty partition is not defined,
-- partitioned Wholes do not share objects,
-- transformations apply to partitions and ensembles,
not to constituent units.
-- all members are identical
-- all ensembles are identical but for the cardinality of parts

Combination of ensembles cannot be interpreted as a type of union operation, since the union of objects from two sets produces a collection of objects, while combining ensembles produces a single united ensemble.

Names and Variables

Although units do not support naming, ensembles do. Later, within an interpretation as integers, we will label ensembles with numerals, {1,2,3,...}. For now, small letters {a,b,c,...} are used as names of particular ensembles.

Boundary and unit calculi utilize ensembles as patterns for substitution and transformation rules. Capital letters {A,B,C,...} are used as variables that range over forms, that is, over Wholes and over partitions. Operations over partitions are rarely mentioned in the sequent, so to keep partitions clearly marked, we will occasionally use italic capitals to refer to partitions. Normal capital letters will be used exclusively to refer to ensembles within a single space.

Here is an example of a partitioned Whole with labels:

A = (●●●|●●●|●●|●)
A = A|B|C|D where A = ●●●, B = ●●●, C = ●●, and D = ●

UNIT AND BOUNDARY ARITHMETICS -- SECTION IIIA October 1, 2008 10:55 PM
William Bricken
January 2007

MEREOLOGICAL CONCEPTS
- Void
  - Void Meta-Notation
- Units
- Fusion
  - Variarity
  - Many-to-One
  - Flatness
  - Pre-associative, Pre-commutative
  - Fusion Relations
    - Axiomatic Structures
    - Self Structures
    - Structures with a Constant
    - Symmetry Structures
    - Ordering Structures
    - Universal Relation
    - Mixing Structures
- Discrete Spaces and Partitions
  - Partition Lattice

MEREOLOGICAL CONCEPTS

Mereology is a formal system not based in set theory that addresses part/whole relations. We now consider several mereological concepts that contribute to a formalization of unit ensembles: void, fusion of parts, partitioning an ensemble, and spatial structure. In general, an ensemble is a mereological Whole with parts that do not support overlap (intersection in set theory).

Void

Mereology studies Wholes and parts of wholes. It does not embrace an empty object, since the empty whole would have no parts. Emptiness cannot participate in an ensemble. When unit parts are present in a space, emptiness is negated, while an empty space, containing nothing, has no parts, no properties, and no mechanism by which it can be identified. That is, an "empty ensemble" does not support even a label. More specifically, empty space is what is contained within an empty boundary.

The syntactic implication of a Void with no name is that all names and labels refer to existent ensembles. When an ensemble ceases to contain parts, its label ceases to exist. Void, the absence of both concept and representation, does however play a significant role in formal unit and boundary arithmetics. It is possible to have a representation of structure with ambiguous reference; this is the conventional notion of a variable. So capital letter variables, those that stand in place of ensembles, can also stand in place of nothing.

Existent structure that denotes Void is said to be void-equivalent. Void-equivalent forms can participate in an equality relation; they are equal to their absence, and thus semantically irrelevant. When a void-equivalence is established, void-equivalent forms can be replaced by their absence, that is, they can be deleted. Formally, such a deletion is a void-substitution.

The equational representation of void-equivalence is non-standard:

«void-equivalent-form» =

It is intentional that nothing is recorded on the right-hand-side of this equation.

Like the empty set, every void-equivalent form is everywhere present within every representation. Unlike the empty set, Void is neither unique nor identifiable. Like the core of Leibniz' argument against replicate universes, identification or perception of Void negates its essential characteristic of being Void.

Void Meta-Notation

We have already referred to Void. The word "Void" is within a meta-vocabulary, we have used it to talk about the concept of absence in boundary mathematics. It is sometimes inconvenient for communication not to be able to incorporate a pointer to what is essentially not there. We do so by inserting a meta-symbol for Void within the equational syntax. To label Void as clearly meta-notation within an equation, we will write the label «void», or >< for short.

===
Void has no properties, including cardinality, so

>< >< is an abuse of meta-notation.
===

The meta-token >< is meant to represent a form that is disappearing into its center. It is always easy to read, since it is a meta-token, it can be deleted wherever it occurs, without loss to meaning. Such a sign is truly a "notation", since it acts as a reminder but does not act as content. Said another way, the meta-token >< is purely cognitive.

Units

Units are solid, in that a unit has no contents other than itself. Units are also empty in that they contain nothing else inside. Solidity of units is a representational aid that conveys the impression that it is not possible for a unit to occupy the space inside another unit. This constraint can be stated as an equation:

• • ≠ •

Boundary mathematics addresses representations with both an inside and an outside. A unit has been defined to have no accessible interior because that particular structure nicely models the units the constitute Numbers. We can also construct a unit boundary that does have an inside:

()

This may be considered to be a Hollow unit. There are two possible configurations of a Hollow unit and its replicate. These simple structures may be represented as configurations of delimiter pairs,

() () and (())

We reserve nested boundaries for a different task, that of depth-value notation. In base-2, a boundary doubles the magnitude of its contents. The corresponding structural equation is

() () = (())

With unit notation, this equation would be written as

• • = (•)

Solid units enforce the accumulation of a magnitude of replicates, permitting an interpretation as whole numbers. Hollow units provide an additional interpretation; we later use them to represent negative units.

It is of passing interest that changing the cardinality rule to an idempotency rule

()() = ()	calling
(()) = ><	crossing

creates a system that is sufficient to fully express propositional logic [refs].

With nesting of Hollow units reserved, we will permit juxtaposition of Solid and Hollow units in the same space. These units are defined to annihilate each other, providing the essential void-equivalence equation for the system of unit ensembles:

• ◊ = ><	annihilation

The two unit rules that govern the construction of an interpretation for integers are:

• • ≠ •	cardinality
• ◊ = ><	annihilation

Nesting-as-doubling will later provide a far more convenient notation for large Numbers.

Fusion

In formal mereology, parts are composed together into Wholes. Each Whole is exactly the sum of its parts, and nothing more. Here, units are indistinguishable parts of wholes; wholes are distinguished by occupying distinct and disjoint spaces. We will borrow from mereology the concept of fusion as a method of construction for ensembles. New ensembles are constructed

by fusion of existent ensembles. For example, the fusion of ●●● with ●● is ●●●●●. Algebraically:

fuse(●●●|●●) = ●●●●●

(●●●|●●) represents two spatially disjoint ensembles, ●●● and ●●. Fusion unites the parts of each ensemble into a single Whole. The parts of the fused Whole are the parts of Wholes that are fused, while the distinction between the Wholes that have been fused is lost. Fusion is not set theoretic, there is no reference to objects, classes, or type hierarchies.

In general,

fuse(a|b) = ab

The label "ab" denotes a single new space containing as parts the ensembles formerly occupying (and defining) spaces a and b. Of course, once fused, the discrete identification of the parts formerly associated with spaces a and b is lost, as are the spaces themselves. Note that "ab" does not denote multiplication, it denotes the ensembles a and b fused into ensemble ab.

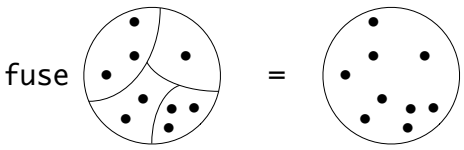
Variarity

For convenience, the above example fuses two spaces. Fusion, however, is blind to the number of spaces being fused. Fusion is simply the simultaneous collection of any number of formerly separate ensembles into a common space. A Whole is simply the fusion of its parts, and in particular, a Whole is not formed by a sequence of "binary" fusions.

An operator that accepts any number of arguments is called variary. Variary functions do not have a specific arity. A more general notation would then be:

fuse(a|...|z) = a...z

An example of the application of fusion in spatial notation:



Typographically, we represent the above spatial equation as

fuse(●●●|●●●|●●|●) = ●●●●●●●●●

Void, the absence of an ensemble, cannot be fused since there is nothing to fuse. Fusion of a single ensemble is an identity:

$$\text{fuse}(a) = a$$

By viewing the parts a,...,z from the perspective of the fused Whole that they form, we can interpret the structure a|...|z as singular, making fusion a unary operation. That is, fusion operates on a singular partitioned form, converting it into a single non-partitioned Whole by deleting existent partitions.

Many-to-One

Fusion maps any partition of a given Whole into that Whole, making fusion a many-to-one spatial operation:

fuse(●●●|●) = fuse(●●|●●) = fuse(●●|●|●) = fuse(●|●|●|●) = ●●●●

Partitions of the same Whole become equal under fusion. Above, the varieties of the partitioned whole ●●●● are semantically equivalent although they differ syntactically. The set of partitions can be arranged into a lattice, with lattice levels counting the number of Wholes constituting a particular partition, That is, the number of parts in the partition. Links between partitions showing available deletions of a single partition barrier.

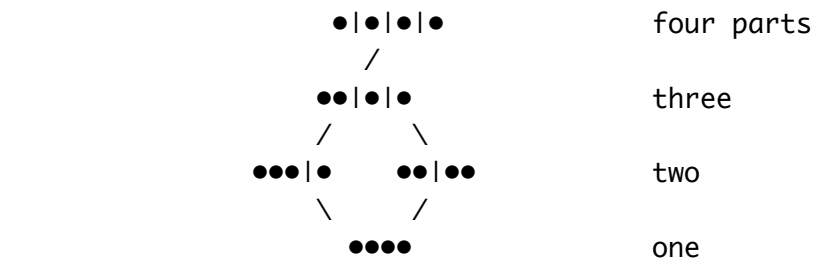


Figure %%%:

The theory of unit ensembles currently incorporates only one sort, that of the ensemble, or Whole, which is composed of units. A Unit Sort is not necessary; in the calculus of unit ensembles, a solitary unit is always an ensemble of one. Besides, due to the indistinguishability of units, there is only one unique member of the unit sort.

Flatness

Fusion converts Wholes in separate spaces into a single Whole in a single space. By construction, ensembles do not form into a hierarchy of structure; fusion enforces this. This mereological operator differs from both set theory and function composition. Specifically, fusion is a "flat" function, any nested applications are subjugated to the top-level fusion application.

fuse(a|fuse(b|fuse(c|d))) = fuse(a|b|c|d) = abcd

fuse(fuse(a|b)|fuse(c|d)) = fuse(a|b|c|d) = abcd

Although we can construct a representation of nested fusions within a token-string representation, nesting evaporates in the more proper spatial syntax. Figure %%% shows that partitions cannot be nested inside other partitions. That is, we do not distinguish ensembles of ensembles.

==========

Figure %%%: Fusion Does Not Require Step-wise Computation

==========

A flat function conforms to the structure

f(f(x)) = f(x)

and more generally

f(u,...,f(v,...),f(w,...,f(x,...))) = f(u,...,v,...,w,...,x...)

The bar notation mimics the flatness of fusion, since it degrades function nesting into identical partitions at the same level

fuse(a,fuse(b,c)) =notation= a|b|c

Flat functions are variary, one step applies the function to any number of arguments. In a resource constrained implementation, arguments may be combined in any groupings and in any hierarchical order, dictated by the convenience of the implementation substrate.

What is strange here, at least from a conventional perspective, is that it appears that temporal composition of fusions is not permitted, that we cannot take steps from a diverse beginning to a singular end. This is exactly what is intended. Temporal and operational steps are treated as occurring in parallel.

In conventional symbol manipulation, we begin with a syntactic structure for which, presumably, we do not know the eventual simplest form. We take steps,

applying a variety of transformations, to eventually terminate with a syntactic structure that is judged to be simple. Solving a set of linear equations, for example, allows us to replace a variable of unknown value by a specific value for that variable. If the solution were known in advance, the steps, presumably, would not be necessary. Consider a simpler example:

1 + 1 + 1 + 1 + 1 + 1 + 1 = 7

In this example, both representations are solutions, depending upon the intended use of the representation. Were the token 7 an iteration variable, we would prefer the decomposed version, in order to make seven iterations. If the particular problem, however, lent itself to computational parallelism (for example, paying for seven penny candies), then we would prefer the non-decomposed representation.

The Additive Principle incorporated within the semantics of unit ensembles assures that the result of taking steps is apparent within the parts prior to taking those steps. Thus there is no motivation to fuse only some parts of a Whole on the way to fusing the entire Whole. Computation is immediate, requiring only one "step", while reading results requires time and effort. The cost imposed by having to count units in a result, via one-to-one correspondence with numerals, is balanced by the benefit of not having to take partial steps toward the result.

We are here seeking to remove the method of implementation and the application specifics from the concept of Number, by formalizing a system that can be interpreted as integer arithmetic and that maintains the intuitive spatial parallelism of the Additive Principle. Thus, should we intend to fuse Wholes a, b, c, and d, as is recorded in the hierarchical token-based representation above, then we need not decompose that fusion into binary steps. We need not impose a step-wise sequential arity on the structure of operations in the spatial formalism. Figure %%% illustrates this by representing fuse(a|b) and fuse(c|d) as inherently incomplete with regard to the Whole that informs the fusion process.

Pre-associative, Pre-commutative

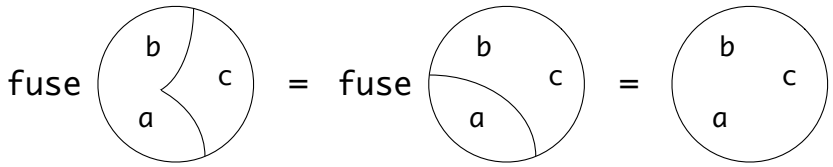
The concept of a flat function disallows several conventional group theoretic relational properties. Although we could write, for example, an associative rule for fusion:

fuse(fuse(a|b)|c) = fuse(a|fuse(b|c))

this would be misleading since the ordering of binary fusions above is an unintended artifact of permitting different partitions of the same Whole to be

distinguished. In spatial notation, the relational structure "associative" is irrelevant to fusion. Figure %%% shows that the associative rule degenerates into a semantically improper choice to partially delete a partition structure.

==========



fuse(ab|c) = fuse(a|bc) = abc

Figure %%%: Associativity is Irrelevant to Spatial Fusion

==========

Associativity conveys a temporal distinction, an artifact of operations that can accommodate only two arguments as once, and therefore require temporal composition. In a spatial model, this distinction is a syntactic implementation detail rather than a feature of the fusion operation. Fusion should be read simply as "delete the separating boundaries",

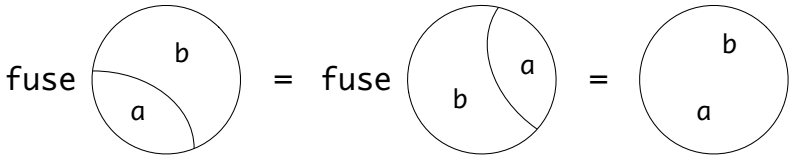
fuse(fuse(a|b)|c) = fuse(a|b|c) = abc

Similarly, a commutativity rule for fusion,

fuse(a|b) = fuse(b|a)

can be constructed only as an artifact of a linearly ordered notation. In spatial notation, the relational structure "commutative" is irrelevant to fusion. Figure %%% shows that the commutative rule degenerates into a syntactic choice of a rotation of a bounded space.

==========



fuse(a|b) = fuse(b|a) = ab

Figure %%%: Commutativity is Irrelevant to Spatial Fusion

==========

The syntactic structure "(a|b)", which is a representation of two disjoint spaces, does not impute a pairwise semantics to fusing parts into a Whole, any more than writing the word "cinq" imparts a French semantics to the concept five. Fusion is essentially a parallel process, independent of both the associative ordering of function application and the commutative ordering of function arguments. We thus call unit ensembles pre-associative and pre-commutative. Ideally the two relational concepts would not be mentioned at all. However, due to the prevalent presumption that integer addition obeys associative and commutative rules of composition, we address the issue directly, by establishing that the theory of unit ensembles is constructed from foundations prior to the conceptualization of either relation.

Fusion Relations

In summary, fusion bears little resemblance to a conventional function. Some of the ways that fusion is non-standard are:

Fusion	Conventional Relation
maps parts onto wholes	maps unique members of a set onto members of another set
accommodates any number of parts	must have specific number of arguments
no position to parts	arguments occupy a specific location
many-to-one	functions are one-to-one
flat, no nesting	can be nested
no concept of associativity	almost always associative
no concept of commutativity	commutative or non-commutative

Modern algebra classifies mathematical systems by the types of structural relations they possess. Well-known structures include

equivalence	reflexive, symmetric, transitive
total ordering	irreflexive, antisymmetric, transitive
partial ordering	reflexive, antisymmetric, transitive

It is somewhat misleading to characterize fusion, and other spatial structures, using relational constructs that require conventional sets and mappings for their definition. We next examine the quasi-relational structure of fusion, overtly recognizing that fusion is not a relation, and that the process of transforming it to conform to relational structures in fact changes it into a mathematical structure that it is not.

Axiomatic Structures

Fusion applies only to ensembles, the "objects" being fused. Ensembles are, of course, not legitimate objects in the sense of unique, atomic, existent set theoretic structures. We address the axiomatic structures of Closure, Existence, and Uniqueness later, within the interpretation of fusion for integer operations. Here we analyze unit ensembles from the perspective of the conventional relational properties for identity.

Self Structures

The relational properties on a single object are Reflexivity Idempotency. The relation of reflexivity, whether or not a relation can be applied to replicates of the same object, is primary. Fusion is not reflexive. The structure that conveys the idea reflexive fusion is

$$fuse(a) = a$$

A space is fused with itself. Reflexivity assumes we can construct a relation that refers twice to the same object:

R ●●●● ●●●●

However, these two arguments are either different ensembles, in which case the proper labeling would be Rxy, or they are the same ensemble, in which case we cannot construct a replicate reference without constructing a new ensemble. Thus, fusion is not only irreflexive, reflexivity itself is not conceptually possible. The deeper issue is the meaning of multiple occurrences of a label for one thing. Free generation of replicate labels is a characteristic of token-string notations, but not necessarily of boundary notations nor of ensembles consisting of indistinguishable units.

Idempotency serves to highlight this issue, since it confers a value on a reflexive structure:

$$Rxx = x \qquad \text{Idempotent}$$

Fusion of ensembles is not idempotent, in that two ensembles can have the same cardinality, yet when they are fused, the value of the resultant form is the value of neither. Rather the fusion has the value of both combined.

$$Rxx = xx$$

That is

fuse(x|x) = xx

The most succinct way to formulate this idea is to apply it to the spatial base case of featureless space: Space itself does not support reference, and most certainly does not support replication of reference. Empty space is not idempotent for an unusual reason. We could write, correctly from a conventional perspective,

R«void»«void» = «void»

However, this equation violates the semantics of Void in two ways. Primarily, empty space cannot be labeled; secondarily, it cannot be labeled twice. We can write the above relation only due to the void meta-notation. In a spatial syntax, empty space is identified only indirectly by an empty container. The empty boundary, (), contains nothing. Generally in a spatial notation, we can refer to space indirectly via boundaries, as in

R()() = ()

Once written, it is easy to see that the equation no longer refers to a relation on empty space, it refers to a relation on containers. More clearly, it may be best just to consider reflexivity to be conceptually irrelevant to the concept of fusion, while idempotency is explicitly denied.

fuse(●|●) ≠ ●

Structures with a Constant

The conventional relations involving a constant are Identity and Invertability.

RxG = x	Identity
Rxx' = G	Invertability

Identity and invertability are the stalwarts of group theoretic structures. Both involve the construction of a ground case that serves to define the relational concept. Identity supposes a ground (or constant) object G that in relation to an arbitrary object x does not change the arbitrary object. Invertability supposes an object x' that in relation to an arbitrary object x, generates the ground object G.

What is the ground object for fusion that creates an identity? Technically it would be the featureless space, since, to distort the concept slightly,

fuse(x|«void») = x

The problem here is that, due to its non-existence, void cannot be fused, and most certainly cannot be construed as a constant object. Within the intended meaning of fusion, there is no fusable ground object.

What then of invertability? Void cannot be implicated in any relational structure, nor can units. But it is certainly possible to have an x' that inverts x to yield void. In the case of units, annihilation of Solid and Hollow units is an example. Voids can be the consequence of a equational operation.

fuse(●|◊) = ● ◊ = «void»

Fusion then is invertable, given the two types of units x and x', which are defined for unit ensembles later as the Solid unit ● and the Hollow unit ◊.

Symmetry Structures

The symmetry relational property includes notions of asymmetry and antisymmetry. Although argument symmetry characterizes fusion, the reason is non-standard. Writing

fuse(a|b) = fuse(b|a)

is wrong. As spatial structures, ensembles are not elaborated sufficiently to possess the concept of a position. Partitions that are fused likewise do not have a relative location that can be used to identify positional symmetry. We might say that partitions are super-symmetric: no matter how you locate them, they appear to be in the same place. This is, of course, stretching the idea. More accurately, with regard to an argument position for fusion, partitions are independent of a symmetry concept. A partitioned whole is a single form that consists of other wholes sharing only one relation, that of being a part of the particular partition.

Ordering Structures

The relational properties that imply an ordering, either temporal or structural, are not relevant to the concept of fusion. These include Associativity and Transitivity. Fusion is associative for the same reason that it is symmetric; the spatial structure of a partition does not support fragmentation into binary (or n-ary) groups for selective transformation. A stronger reason for the irrelevance of associativity to fusion is that it is a flat function, partitions are fused concurrently. From the perspective of the abstract fusion operation,

there is only one argument, the partition to be fused. This perspective converts fusion into a monadic function.

Fusion is intransitive in that the fused structure a|b and the fused structure b|c do not imply that we have a fused structure a|c. Figure %%% shows this.

==========
Fig. %%%:
==========

Transitivity creates an interpretative problem in the context of unique referents. What do the logical structures AND and IMPLIES mean with regard to a flat function operating on unique ensembles? Consider the spatial representation in Figure %%%. Using symmetry irrelevance, we have a case in which two different ensembles (a and c) are fused to a third (b). Flat fusion might interpret the AND and yet another fusion

fuse(a|b) AND fuse(b|c) == fuse(fuse(a|b)|fuse(b|c)) = fuse(a|b|b|c) = abbc

The difficulty with this interpretation is that ensemble b is fused twice into the result. The conjunction AND could mean not sharing a space, but simply existing in two separate spaces:

fuse(a|b) fuse(b|c)

This places the burden of interpretation on the dual location of the ensemble b. And as we have seen, transitivity is not intended to mean "two separate ensembles identical to b". We could defer to a Venn diagram-like form, shown in Figure %%%, interpreting AND as spatial identity of arguments.

==========
Fig. %%%:
==========

Then most certainly fusion would be transitive, or rather super-transitive, since all parts fused in pairs would immediately be parts of the same flat fusion. But then the implication that fuse(a|c) is the case becomes suspect, since we have absolutely no mechanism to hold off the fusion of a and c so that it can be an implication of two existent fusions a|b and b|c. That is, conjunction and implication degrade into the same thing.

This makes the property of transitivity particularly interesting, since the logical connectives appear to invite a temporal construction. Should one have to take a step to construct a|b and b|c and then a following step to conclude that a|c has been constructed? This interpretation certainly does not characterize a flat function, nor the non-temporal semantics of logic.

"Flat transitivity" might better be written as

fuse(a|b) AND fuse(b|c) = fuse(a|b|c)

However this very special interpretation is in essence saying that fusion is idempotent, that replicates do not fuse, leading us far afield from the original spatial concepts.

What does transposition mean for a conventional relation? It is not temporal, but observational. Antecedents a|b and b|c are current observations that, should we look, are also accompanied by another observation, a|c. This perspective makes fusion non-transitive, since the two fusions imply nothing about any other fusions. The map between the ensembles a and b and their fusion ab is independent of the map between ensembles b and c and bc, and it is independent of the map between ensembles a and c and the fusion ac. Transitivity, like commutativity and associativity, is irrelevant.

Universal Relation

===
--conclude universal but for reflex. Everything connected/transforms/ symmetrical except identity.--
===

Another way to look at fusion is that it is most similar to a universal relation. Universal relations are true of every possible combination of objects. But universal relations convey nothing about particular structure; they are more appropriately seen as properties of the entire system than of a particular operator.

Every possible way of combining ensembles, from 2 to N at a time, generates a new, but not necessarily unique, ensemble. Fusion is, in a conventional vocabulary, a set-based unary function, taking all members of the power set of ensembles into the set of ensembles.

It seems best just to say, also, that transitivity is irrelevant to a flat function such as fusion. Transitivity is also irrelevant to the addition of integers, since addition is a ternary relation.

==========

Fig

(a|b) (b|c) (a|c)

(a| b |c)

==========

Mixing Structures

Distribution of fusion over substitution, the other transformation within the system of unit ensembles, will be considered in Section %%%, after a discussion of substitution itself.

Where has this exploration lead us? Conventionally, fusion is an invertable function with no identity that distributes over substitution. The rest of the properties of conventional relations appear not to apply. And where then does fusion fit into the theory of groups? Here we may try adding in the irrelevant properties of irreflexivity, asymmetry, associativity, and transitivity. Doing so brings us to a semigroup that would be a group were it not for the non-existence of void. It is surely best just to accept that fusion is not group theoretic.

Figure: Properties of Relations

Axiomatic Structures		
Closure		
Existence		
Uniqueness		
Self-Structures		
Reflexive/Irreflexive	Rxx	NOT
Idempotent	Rxx = x	
Structures with a Constant		
Identity	RxG = x	
Invertable	Rxx' = G	
Symmetry Structures		
Symmetric/Asymmetric	Rxy = Ryx	
Antisymmetric	Rxy and Ryx implies x=y	
Ordering Structures		
Associative	Rx(Ryz) = R(Rxy)z	
Transitive	Rxy and Ryz implies Rxz	NOT
Mixing Structures		
Distributive	R(Sxy)(Szw) = S(Rxy)(Rzw)	

Fig Summary of Fusion Quasi-relations

Quasi-relation	Reason
Self	
irreflexive	irrelevant due to uniqueness of representation
not idempotent	irrelevant due to uniqueness of representation
Constant	
no identity	no ground structure
invertable	annihilation is permitted
Symmetry	
symmetric	irrelevant due to spatial freedom
asymmetric	inside/outside spatial location
antisymmetry	no
Ordering	
associative	irrelevant due to spatial freedom
transitive	irrelevant due to flatness
Mixing	
distributive	over put/in substitution forms, but not for-forms

Discrete Spaces and Partitions

We mention the partition operation only briefly, since it is not used in our formalization of unit arithmetic. Partition can be added to extend the theory of unit ensembles by differentiating between structurally varying partitions of the same Whole.

A single partition consists of unique labeled spaces, each containing a single ensemble. Here, partitioning is more limited than general mereological decomposition, since by definition, all unit replicates within a space and all ensembles are disjoint. Partitions are collections of occupied spaces.

Fusion can be seen as the inverse of a partition operation,

a...z ==> (a|...|z) PARTITION

which separates or divides a Whole into constituent parts.

Partitions is a unary operation that maps a Whole onto a set of possible partitions; it is therefore one-to-many. For example:

partitions(●●●●●) ==>
{(●●●●|●), (●●●|●●), (●●●|●|●), (●●|●●|●), (●●|●|●|●), (●|●|●|●|●)}

Using a single partition generator,

partition(a) = choose-one-of(partitions(a))

we can express the inverse relation between fusion and partition:

fuse(partition(a)) = a

partition(fuse(a|...|z)) = a|...|z

In a maximal partitioning, all constructed spaces containing exactly one atomic unit, so that the maximal partition consists of named spaces in one-to-one correspondence to units in the partitioned ensemble. For example:

max-partition(●●●●●) = (●|●|●|●|●) = (a|b|c|d|e) where a = b = c = d = e = ●

The conventional counting operation usually depends upon a maximal partition, although we can also count "by twos" or by groups of another magnitude. Unit arithmetic is formulated to avoid unit iteration in favor of direct one-to-one correspondence, and to avoid sequential iteration in favor of parallel operations.

==========

Here are all the partitions of the five smallest ensembles:

Whole	Partitions
●	none
●●	(●|●)
●●●	(●●|●), (●|●|●)
●●●●	(●●●|●), (●●|●●), (●●|●|●), (●|●|●|●)
●●●●●	(●●●●|●), (●●●|●●), (●●●|●|●), (●●|●●|●), (●●|●|●|●), (●|●|●|●|●)

==========

Note that a partition must consist of two or more parts.

Partition Lattice

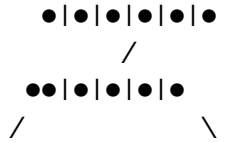
Partitions can be organized into a (semi?)lattice, with no separating boundaries within the minimal element, and N separations within the maximal element. The minimal element is the Whole, without partitioning, while the maximal element has a separating boundary around every unit that constitutes the whole. Thus there are N-! boundaries in the maximal element of the lattice for the Whole N.

The levels of the lattice count the number of separations within a form, so there are N levels (with partitions ranging from 0 to N-1 separations). If the partition function is viewed as strictly binary, then the lattice identifies steps that generate each possible partition, each binary-partition step adds a single separator.

A partitioning transformation that permits only one partition per application might be characterized as a binary function modeled by the lattice Join (&&&check) operation. An iterative sequential mapping of steps that add or delete exactly one separator permits traversal of the lattice.
However, in a more parallel fashion, the lattice can be treated as a single object, with a variary function that maps to all possible partitions (i.e. a lattice-function). Partition is such a function, mapping a Whole representing the integer N onto a lattice of possible partitions of N. Fuse is the many-to-one inverse function that maps any lattice element onto the corresponding Whole.

For example, the lattice for ●●●●●● is displayed in Fig %%%:

==========



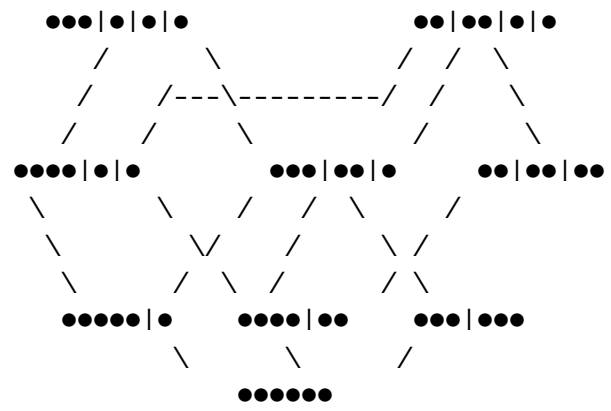


Fig %%% Layers: 1-1-2-3-3-1 = 11
==========

Since fusion lattices are so densely interconnected, the anti-lattice of non-connections is preferable. The anti-lattice of non-connections for ●●●●●● are in Figure %%%:

==========

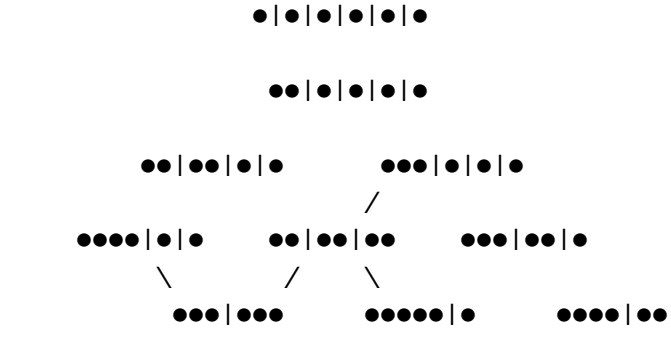


Fig %%% Layers: 1-1-2-3-3-1 = 11
==========

In summary, we have constructed a limited mereological theory of unit ensembles that does not support overlap, or intersection, of parts. Wholes are composed of indistinguishable atomic units. Spatial forms are either Wholes or partitioned Wholes. From a conventional perspective, we currently have a formal theory with one Unit, one Sort, and one variary, many-to-one Fusion operation, as is shown in Fig %%%. The definition of a form as either a unit or a fusion of wholes provides sufficient mechanism for decomposition of wholes.

===
Partitions

A number can be partitioned into subnumbers by drawing a distinction between the components of the number. This defines addition.

eg

•	as	•/						
••	as	••/	•/•					
•••	as	•••/	••/•	•/•/•				
••••	as	••••/	•••/•	••/••	••/•/•	•/•/•/•		
•••••	as	•••••/	••••/•	•••/••	•••/•/•	••/••/•	••/•/•/•	•/•/•/•/•
••••••	as	••••••/	•••••/•	••••/••	•••/•••	••••/•/•	•••/••/•	••/••/••
				•••/•/•/•	••/••/•/•	••/•/•/•/•	•/•/•/•/•/•	

A subset of these subnumbers defines multiplication, wherein all subdivisions are equal.

•	as	•/			
••	as	••/	•/•		
•••	as	•••/	•/•/•		
••••	as	••••/	•/•/•/•	••/••	
•••••	as	•••••/	•/•/•/•/•		
••••••	as	••••••/	•/•/•/•/•/•	•••/•••	••/••/••
•••••••		•••••••/	•/•/•/•/•/•/•		
••••••••		••••••••/	•/•/•/•/•/•/•/•	••/••/••/••	••••/••••

●●●●●●●●● ●●●●●●●●●/ ●/●/●/●/●/●/●/●/● ●●●/●●●/●●●

A subsets of these subnumbers defines exponentiation, wherein all subdivisions are equal, and the number of subdivisions also equals the subdivision number.

● as ●/

●●●● as ●●/●●

●●●●●●●●● ●●●/●●●/●●●

8 = 2^3

●●●●●●●● ●●/●● / ●●/●●
===

===
Equivalences

An equivalence set is a relation which is

reflexive	xRx
symmetric	xRy -> yRx
transitive	xRy and yRz -> xRz

Partitions

A partition of a set (or a relation) is a collection of disjoint subsets of the set. The union of partitions is the entire set.

The equivalence relation determines a partition, and each partition of a set defines an equivalence relation.
===

==========

Figure:

W(•)	*a* single unit is *a* Whole
when W(*a*) and W(*b*), W(fuse(*a*|*b*))	fused ensembles are Wholes
fuse(*a*) = *a*	identity
fuse(*a*|...|*z*) = *a*...*z*	unary many-to-one relation

This permits *a* constructive characterization of Wholes:

when W(*a*), W(*b*) and W(*c*), *a* = • or *a* = fuse(*b*|*c*)

==========

We now construct *a* theory of equality for fused Wholes.

UNIT AND BOUNDARY ARITHMETICS -- SECTION IIIB October 1, 2008 10:55 PM
William Bricken
January 2007

EQUALITY AND SUBSTITUTION

Algebra is the formal manipulation of abstract symbols that themselves may be interpreted widely. The primary connective structure in an algebra is equality, which defines an equivalence relation on the syntactic structures of a system. Structures with the same intent, or value, are equal, even though they may be different structures. A simple example is the equation

$$1 + 1 = 2$$

Here the expression 1+1 differs structurally from the expression 2, yet both are equal in numerical value.

Formal transformation means following explicit and unambiguous structural transformation rules. Substitution and replacement of equivalent symbolic structures are deep properties of algebraic equality. An algebraic computation can be characterized as substitution of equivalent syntactic patterns.

The boundary mathematics generalization of syntactic structure to two and more dimensions can be incorporated within a conventional theory of algebraic equality by identifying equivalence classes of spatial forms. Valid transformations are identified by algebraic rules that define equivalent spatial patterns. Equivalent patterns can be substituted for one another without undermining an assertion of equality.

Axioms of Equality

Equations specify valid transformations. They also obey several relational constraints, due simply to the nature of equations. Without regard to interpretation, the conventional relational constraints on the equality relation hold:

$A = A$	reflexive
$A = B$ iff $B = A$	symmetric
$A = B$ and $B = C$ iff $A = C$	transitive

When an equation asserts the two spatial structures are equal, the same rules hold. Expressing a formalization in equations is particularly handy, since it permits syntactic identity. Forms that appear the same are identical. Forms that are equal in value can be transformed into one another by syntactic substitution rules.

Functional Substitution

Functional substitution is a property of equality. When two forms are equal, the applications of a function to each form are also equal:

```
      forall A,B  if A=B then F(A) = F(B)
```

More generally,

```
      forall A,B  if A=B then
                    F(x1,...xi-1,A,xi+1,...,xn) = F(x1,...xi-1,B,xi+1,...,xn)
```

===
add Leibniz LAw
===

Logical Inference

--ifthen use iff

===
Since we favor an entirely algebraic notation, we define implicative relations to be bidirectional, rather than axiomatizing one directional implication and then demonstrating the other direction.
===

Pattern Substitution

The general rules of pattern substitution apply within an equational theory. In pattern substitution, all occurrences of a "for-form" within an "in-form" are replaced by a "put-form":

sub(<put-form>, <for-form>, <in-form>) ==> <new-in-form>

This would be read: "Substitute the pattern <put-form> for each literal occurrence of the pattern <for-form> within the pattern environment defined by <in-form>."

Here we introduce the simpler notion of replacing any occurrences of the for-form within the in-form by the put-form, whether or not the put-form and the for-form are equivalent.

Transforming an Environment

The in-form defines a context for the scope of a substitution. The in-form is the environment that is changed by a substitution. When the put-form is exchanged for an equivalent for-form, the substitution will alter the particular structure of the environment, but it will not alter its value.

We focus first on the general capability of changing an environment by following specified pattern substitutions, without concern for value. Some substitutions maintain equality across portions of an environment but do not maintain the value of each portion. Some substitutions degrade into trivial base cases. Some change the value, or meaning, of the environment, others do not. In this Section, we identify the structural properties of substitutions in general, using unit ensembles as an example. We do not yet employ an interpretation as integers, and so do not introduce addition or multiplication. Just like fusion can be interpreted as addition, we will later show that substitution can be interpreted as multiplication.

Types of Valid Substitution

A valid substitution does not alter the value of the environment, the value prior to substitution remains invariant after substitution.

The primary method of algebraic transformation is substitution in which the put-form is equivalent to the for-form. This is commonly known as "substitution of equals for equals". We discuss this type of substitution later, as value maintenance.

A second type of substitution invariance occurs whenever the environment is an equation. In this case the equivalence of each side of the in-form equation is maintained during substitution even when the put-form is not equal to the for-form. This type is called "global replacement", since all occurrences of the for-form must be replaced by the put-form in order for in-form equivalence to be maintained.

A third type of substitution invariance occurs when we apply the equational rules of functional substitution to substitution itself. Replacing equals for equals in the put-form, or in the for-form, or in the in-form, constructs new substitution patterns that are equivalent to those of the original substitution pattern.

Thus there are three different ways that the value of form can be maintained during substitution. In substitution of equals, value is maintained because the forms being exchanged are equal. This is semantic substitution, since the put-form is a different (but equal) syntactic structure than the for-form. Global replacement of a for-form in an equation for an arbitrary put-form, however, is syntactic substitution, since the replacement relies upon the equality within the equational in-form. Finally functional substitution is structural, the structure of a substitution pattern is converted into a different but equivalent substitution pattern.

We next provide an example of each of the types of substitution within the context of unit ensembles.

Substitution of Equals

Fusion does not change the number of units within an ensemble. With regard to the number of units, we can assert this as

fuse(a|b|c) = abc

As a specific example,

fuse(●●●|●●|●) = ●●●●●●

Substitution of equals applies to partitions as well as to wholes:

sub(x, (a|b|c), (a|b|c|d|e)) = (x|d|e)

We can substitute a partitioned form for a Whole:

sub((a|c), (b|c|d), (a|b|c|d|e)) = (a|a|c|e)

Because we are substituting an equal for an equal, it is not necessary to make every possible substitution. That is, substitution does not need to be global.

Global Replacement within an Equation

The equality expressed by an equation is maintained whenever one form is globally replaced within the equation by another form, regardless of whether or not the two forms are equal. Another way of saying this is that substitution distributes over equality.

sub(a,c,e=f) = sub(a,c,e)=sub(a,c,f)

As a specific example,

sub(●●,●, fuse(●●|●)=●●●) = sub(●●,●,fuse(●●|●))=sub(●●,●,●●●)

sub(●●,●, fuse(●●|●)=●●●)
= fuse(●●●●|●●)
= ●●●●●●

sub(●●,●,fuse(●●|●))=sub(●●,●,●●●)
= fuse(●●●●|●●)
= ●●●●●●

Although the unit value of the in-form fusion equation is changed on both sides of the equation, the assertion of equality is maintained.

Functional Substitution Applied to Substitution

Applying functional substitution to each of the arguments of the substitution operation yields three structural rules for valid substitutions:

if A=B then	sub(A,C,E) = sub(B,C,E)	put-equality
if C=D then	sub(A,C,E) = sub(A,D,E)	for-equality
if E=F then	sub(A,C,E) = sub(A,C,F)	in-equality

For the application to unit ensembles, the relation between ensembles and equality of substitution is bidirectional, allowing us to write the equality rules as equivalences: The three substitution equality rules are:

A=B	iff	sub(A,C,E) = sub(B,C,E)	put-equality
C=D	iff	sub(A,C,E) = sub(A,D,E)	for-equality
E=F	iff	sub(A,C,E) = sub(A,C,F)	in-equality

```
================================================================================
Example:
      A = 2+3    B = 3+1+1   C = 3    D = 2+1    E = 5+3+2+1    F = 4+3+2+1+1

      sub(2+3, 3, 5+3+2+1)  =  sub(3+1+1, 3, 5+3+2+1)

            5+(2+3)+2+1     =     5+(3+1+1)+2+1

      sub(2+3, 3, 5+3+2+1)  =  sub(2+3, 2+1, 5+3+2+1)

            5+(2+3)+2+1     =     5+3+(2+3)

      sub(2+3, 3, 5+3+2+1)  =  sub(2+3, 3, 4+3+2+1+1)

            5+(2+3)+2+1     =     4+(2+3)+2+1+1
================================================================================
```

Condensed Notation

We introduce a more elegant notation that allows axioms to be stated clearly and that makes algebraic proofs easier both to construct and to follow. Unit ensembles already have the advantage of an algebraic formulation, reducing transformation sequences to well understood steps of substitution of equals for equals. We have also eliminated logical implication, if...then..., in favor of bidirectional implication, iff, the statement of logical equivalence.

In pursuit of simplicity, we have also limited the theory of unit ensembles to only two operations, variary fusion and ternary substitution. The formulation of axioms and theorems incorporates principles of boundary mathematics, for which spatial enclosure indicates relational structures. We can now abbreviate the notations for substitution and for fusion.

Substitution requires three ordered arguments. Substitution is represented by unlabeled square brackets. Thus we can take as implicit that forms enclosed in unlabeled square brackets are substitution triples. For further simplicity, we delete the commas that separate arguments:

sub(a,c,e) =notation= [a c e]

This use of square brackets is an example of a boundary notation. In boundary notation, an operation is named by a delimiting boundary rather than by a token.

A partition is indicated by an outer boundary and inner separation bars, such as in (a|b|c). For a shorthand notation for fusion, we will delete the outer boundary. The notation shows the intended fusion by an implicit boundary, the same type of implicit boundary used for wholes.

fuse(a|b|c) =notation= a|b|c

The act of fusion converts the notation above into a fused whole by removing the separating bars,

a|b|c = abc

Thus the notation (a|b|c) is a partition containing different spaces, the notation a|b|c is an instruction to fuse the contents by removing bars, and the notation abc represents the result of the fusion instruction.

We continue to explicitly write sub(...,...,...) and fuse(...|...|...) when emphasis is desired.

Special Cases

As well as the three types of substitution invariance, there are special cases that often permit substitutions to be simplified. We consider total substitutions and void substitutions.

Self and Universe

Substitution rules include two specific base cases that are justified by the definition of substitution itself:

[A E E] = A	global substitution
[A A E] = E	self substitution

Global substitution replaces the entire in-form by the put-form. It is a maximal substitution, a replacement of the universe from the context of E.

Self substitution replaces the put-form by a for-form that is an identical replicate. Since the replicate is indistinguishable from the original, self substitution is the null substitution operation.

Void Substitution

Void-substitution (i.e. substitution of void for a given non-void form) is primary method of spatial form reduction via deletion:

[>< C E] = delete(C,E) put-void substitution

"Delete(C,E)" means to delete all literal occurrences of the pattern C within the form E. Deletion of a form C will usually change the value of form E. The only case when it would not change the value is when the put-form is equal to the for-form. In the case of void-substitution,

[>< C E] = E iff C = ><

Formally, deletion is void-substitution, substituting nothing for a syntactic something. Informally we say that form C is erased. The value of E is maintained only when a deleted form is void-equivalent.

For example, let us preview a void-equivalent equation that will be explained later:

- ◊ = ><

Consider this equation to be a pattern, without interpretation. An example of void-substitution is:

[>< ●◊ ●●◊●◊●] = ●●

Substituting an existent put-form for a void for-form is also valid, since this is void-substitution in the constructive direction. In a calculus which permits some forms to be equated with void, those void-equivalent forms can be constructed in place of Void (i.e. anywhere) in an expression.

[A >< E] = construct(A,E) for-void substitution

Using the void-equivalent equation above, here are several equivalent substitutions:

[●◊ >< ●●] = ●●◊● = ●◊●●●◊ = ●●◊●◊● = ●◊●◊●◊●◊●◊●◊●●

One obvious subtlety with void-substitution is that void-equivalent forms can be inserted without limit, in any space within or outside of E. This is never a difficulty, since the choice to insert a void-equivalent form can be made solely on pragmatic grounds. Void-equivalent forms are formally irrelevant to value, or meaning, in that construction of a void-equivalent form in E does not change the value of E. Thus void-equivalents can be constructed anywhere, but they are constructed only where they are desired.

The only way to modify a void in-form by substitution is when the for-form is also void. In this case only, the for-form will match the in-form, resulting in substitution of the put-form:

[A C ><] = >< in-void substitution

But

[A >< ><] = A global void substitution

This type of substitution provides a base case of global substitution. For example,

[●◊ >< ><] = ●◊

Value Maintenance

We have defined above a general concept of substitution. Substitution must be restricted whenever the value of the in-form is to be maintained. This important constraint permits substitution to be used freely within an algebra.

We are usually interested only in maintaining values during substitution. Thus a general restriction of the use of substitution is that the put-form is equivalent to the for-form. This equivalence is commonly stated in the form of an equation that serves as a rule or a pattern substitution template.

```
[A C E] = E  iff  A = C            value maintenance
```

Substitution of equals for equals maintains the value of the in-form, although the syntactic structure is changed. For example:

```
================================================================================
Fig. %%%:  a = three in a spatial form, c = three also, put each into E
================================================================================
```

Self substitution is the base case for value maintenance. The value maintenance constraint further restricts void-substitution to permit only those modifications of the in-form that maintain value. In particular,

```
[>< C E] = E  iff  C = ><
[A >< E] = E  iff  A = ><
```

An issue is whether or not a substitution is possible at all. When the for-form is not contained somewhere within the in-form, a substitution is not possible. This is called "failure to match". For the application to unit ensembles, we will assure that failure to match never occurs.

Global Replacement

Global replacement of a form for another maintains the value of an in-form that asserts an equality.

```
[A C E=F]  =  [A C E]=[A C F]            distribution
```

The substitution of A for C distributes over the assertion of equality. This conventional rule for term rewriting bears a strong resemblance to functional substitution over in-forms:

```
[A C E] = [A C F]  iff  E = F            in-equality
```

===finish
We see hear a link between Leibniz' rule of functional substitution, also known s %%%, and the predicate calculus inference rule of replacement
===

Associativity of Substitution

For composed operations of one type, associativity gives permission to perform these operations in any order. It is commonly expressed as

$$a \odot (b \odot c) = (a \odot b) \odot c$$

where the circled dot, $\odot$, stands in place of any associative binary operator.

Substitutions can be composed in several distinct ways. The first set of substitutions below apply the substitution of b for d to another substitution, and then to each of the arguments of that substitution.

```
[b d [a c e]]          outer-composition
[[b d a] c e]          put-composition
[a [b d c] e]          for-composition
[a c [b d e]]          in-composition
```

This, and other varieties that apply parts of a substitution function to another substitution, fall broadly under the rule of functional substitution in equations. In the next set, the arguments of an outer substitution are sequentially composed in groups of three. If we assume that each of the labeled forms are disjoint, then each group of substitutions is an equivalence set.

```
[[a c b] d e]          put-first
[a [c b d] e]          for-first
[a c [b d e]]          in-first
```

For our application to unit ensembles, we will see that all of these forms of associativity of substitution are equivalent.

Distribution of Fusion over Substitution

Fusion combines the contents of multiple spaces into the singular contents of a single space. Since the parts of a fusion are disjoint, fusing substituted parts into a whole is the same as substituting the fused whole in the first place. That is,

```
[a|b c e] = [a c e]|[b c e]
[a c e|f] = [a c e]|[a c f]
```

Since the put-form and the in-form are independent, the two varieties of distribution above can be combined into a single rule:

[a|b c e|f] = [a c e]|[b c e]|[a c f]|[b c f]

Here is an example of distribution of fusion over substitution. Let

a = • b = •••
c = •• d = ••
e = •• f = ••••

[a|b c e] = [•|••• •• ••] = [•••• •• ••] = ••••
[a c e]|[b c e] = [• •• ••]|[••• •• ••] = •|••• = ••••

[a c e|f] = [• •• ••|••••] = [• •• ••••••] = •••
[a c e]|[a c f] = [• •• ••]|[• •• ••••] = •|•• = •••

[a|b c e|f] = [•|••• •• ••|••••] = [•••• •• ••••••] = ••••••••••••
[a c e]|[b c e]|[a c f]|[b c f]
= [• •• ••]|[••• •• ••]|[• •• ••••]|[••• •• ••••]
= •|•••|••|•••••• = ••••••••••••

Note that

[a c|d f] ≠ [a c f]|[a d f]

since the for-form must be treated as a single Whole. We demonstrate failure of for-form distribution using the same values as above,

[a c|d f] = [• ••|•• ••••] = [• •••• ••••] = •
[a c f]|[a d f] = [• •• ••••]|[• •• ••••] = ••|•• = ••••

Another restriction is that distributed substitution cannot be applied partially:

[a c e|f] ≠ e|[a c f]

since the substitution of a for c can change the value of e|f. For example, using the values above:

[a c e|f] = [• •• ••|••••] = [• •• ••••••] = •••
e|[a c f] = ••|[• •• ••••] = ••|•• = ••••

Substitution Relations

We now consider the conventional properties of relations as applied to substitution. Since substitution is a three place operator, we will consider a simpler version for which the put-form is not a parameter, but rather held constant.

substitute-put(<for-form>, <in-form>)

We will, however, continue to use the three argument condensed notation for consistency.

```
================================================================================
Axiomatic Structures                ---FIX---
      Closure
      Existence
      Uniqueness
Self-Structures
  Reflexive/Irreflexive       Rxx          [P x x]           yes
  Idempotent                  Rxx = x      [P x x] = P       no not =x
  Identity                    RxG = x      [P x g] =?= x     G=E
  Invertable                  Rxx' = G     [P x x'] =?= E    x'=x
Symmetry Structures
  Symmetric/Asymmetric        Rxy = Ryx          [x y E] =?= [y x E]
  §Antisymmetric        Rxy and Ryx --> x=y      [x y E] and [y x E] -?-> x=y
Ordering Structures
  Associative (NESTING) Rx(Ryz) = R(Rxy)z        [x [y z E] E] =?= [[x y E] z E]
  §Transitive           Rxy and Ryz --> Rxz      [x y E] and [y z E] -?-> [x z E]
Mixing Structures
  Distributive          R(Sxy)(Szw) = S(Rxy)(Rzw)
                              [x|y z|w E] =?= [x y E]|[z w E]
================================================================================
```

what is identity? inverse?

where does this leave us? subinE G=E x'=x antisym assoc-as-nesting

```
================================================================================
SUMMARY of SUBSTITUTION RULES  (extended notation)

      if A=B then  sub(A,C,E) = sub(B,C,E)           put-equality
      if C=D then  sub(A,C,E) = sub(A,D,E)           for-equality
      if E=F then  sub(A,C,E) = sub(A,C,F)           in-equality

      sub(A,E,E) = A                                 global substitution
      sub(A,A,E) = E                                 self substitution

      sub(A,C,E) = E   iff  A = C                    value maintenance

      sub(A,C,E=F)  =  sub(A,C,E)=sub(A,C,F)         distribution over equality

      sub(«void»,C,E) = delete(C,E)                  put-void substitution
      sub(A,«void»,E) = construct(A,E)               for-void substitution
      sub(A,C,«void») = «void»                       in-void substitution
================================================================================

================================================================================
SUMMARY of SUBSTITUTION RULES  (condensed notation)

      [A C E] = [B C E]  iff  A = B            put-equality
      [A C E] = [A D E]  iff  C = D            for-equality
      [A C E] = [A C F]  iff  E = F            in-equality

      [A E E] = A                              global substitution
      [A A E] = E                              self substitution

      [A C E] = E  iff  A = C                  value maintenance

      [A C E=F]  =  [A C E]=[A C F]            distribution over equality

      [>< C E] = delete(C,E)                   put-void substitution
      [A >< E] = construct(A,E)                for-void substitution
      [A C ><] = ><                            in-void substitution
================================================================================
```

WHOLES AND HOLES

In preparation for mapping to integers, we introduce a second type of unit, ◊, that annihilates the unit ●. In a spatial calculus, unit annihilation results in void. Later we will interpret diamond, ◊, as -1.

fuse(●|◊) = ● ◊ = void Annihilation

Two Sorts

By adding the new type of unit, we also add two new sorts. We keep the W sort to describe ensembles composed of either ● or ◊ units, and specialize new sorts for ensembles that are composed entirely of one type of unit only.

S(●)	Solid unit
H(◊)	Hollow unit
S(a)	Solid ensemble
H(a)	Hollow ensemble

An ensemble is Solid, or S, if it consists only of S units. Similarly, an ensemble is Hollow, or H, if it consists only of H units. Later, we will assign the property P, for Positive, to be an interpretation of S, and the property N, for Negative, to be an interpretation of H.

Note that void does not have a Sort, since Void has no properties. There is sometimes a philosophical confusion that "the property of having no properties is itself a property". This comes about only from an initial error of assigning the property of existence to void. By accepting that Void does not exist, this confusion reduces to: "the property of not existing is itself a property". Such a position has two major flaws. First, we would then be in a position of having to assign properties to all non-existent things (thus bringing the inconceivable into existence), and second, Void does not support identification, so we would be in the awkward position of assigning a property to something that we cannot refer to.

Void-based Models

The annihilation rule is analogous to the creation of particles and antiparticles in atomic physics. Pairs of units and anti-units arise out of potentia and return to nothingness. Structure is what happens in between.

Unit annihilation permits a calculus for which unit value is maintained at "zero", here at non-existence. This type of calculus does not permit direct construction of units. Direct construction would require a permission such as

$$\bullet = \text{«void»}$$

This permission in fact undermines the idea of value. Instead, all structure is brought into existence from Void, value is maintained at Void, and when all structure is fused, unit annihilation would return any representation to void.

A system with only two sorts has Boolean characteristics. However, annihilation reduces replicates of the two unit types to a third possibility, that of having no sort at all (i.e. void). Here is an example of a void-based rule that extends the two-sorted system beyond the two sorts. We have in effect, values for Solid and Hollow units, and implicit values for absence of units.

No Coexistence

Ensembles now come in three varieties: Pure Solid, Pure Hollow, and a mixture of S and H units. However, due to the unit annihilation rule, a mixture of Solid and Hollow units is unstable in that it can exist only prior to reduction via Unit Annihilation. We will not need a Mixed Sort so long as we include Unit Annihilation within an extended concept of fusion. Thus, to fuse ensembles, we will place their parts in the same space and apply Unit Annihilation. Formally then an ensemble will always be pure.

$$\begin{aligned} W(a) &\quad \text{iff} \quad S(a) \text{ xor } H(a) \\ W(a|b) &\quad \text{iff} \quad W(a) \text{ and } W(b) \end{aligned}$$

We use the shorthand notation $\bullet a$ for Solid a, and $\lozenge a$ for Hollow a, and a non-prescripted a for cases when the type of a is unknown or irrelevant. Thus

$$\begin{aligned} \bullet a &\quad \text{iff} \quad S(a) \\ \lozenge a &\quad \text{iff} \quad H(a) \\ a &\quad \text{iff} \quad W(a) \end{aligned}$$

ChangeUnit

We introduce a specialization of substitution for which every unit of one type is exchanged for a unit of the other type.

$$\text{changeUnit}(E) \quad =\text{def}= \quad \text{in parallel, } \text{sub}(\lozenge,\bullet,E) \quad \text{and} \quad \text{sub}(\bullet,\lozenge,E)$$

Let us abbreviate concurrent changing of units as

ΔE =def= [●/◊ ◊/● E]

The diagonal slash is shorthand for applying two different substitutions at the same time.

The base cases of changeUnit are

Δ● = ◊
Δ◊ = ●

An example application is

Δ●●●◊◊●◊ = ◊◊◊●●◊●

ChangeUnit does not maintain the value of E, since it converts E into its inverse. Thus,

ΔE = E iff E = ><

Under an interpretation for integers, changeUnit does maintain the absolute value of an ensemble. At this point, we have not yet introduced the concept of magnitude, so that changeUnit can be seen purely as a type changing operation.

We can say that ΔE is the inverse of E with respect to Solid and Hollow units. It will become the Additive Inverse when interpreted for integers. However, under the integer interpretation, changeUnit is also the "Subtractive Inverse". By introducing two distinct types of units, we have unified addition and subtraction into the single concept of fusion.

With exactly two types of units, changeUnit is its own inverse:

ΔΔE = E inverse changeUnit

Since changeUnit is an application of substitution, it distributes over fusion. In condensed notation,

Δ.A|B = ΔA|ΔB

The dot indicates that the scope of changeUnit extends over the entire fusion.

Identity

Fusing a Whole with its inverse results in void:

a|Δa = ><

Consider any unit in a. Every unit of either type will be fused with its inverse unit to form a ●◊ pair which will then annihilate into void. This is an example of a parallel rather than an iterative execution of proof. We do not need to iterate through each ●◊ pair, since every pair annihilates concurrently. All that is needed for a proof is the one-to-one correspondence between ● and ◊ units. This correspondence is given by the definition of Δa.

Parallel annihilation is assured because units have no relational dependencies. The parallelism is deeper still, since immediately after fusion, Solid and Hollow units are not even coupled in pairs. Any ● can annihilate any ◊. This type of concurrency is molecular. An implementation analogy is precipitation of a salt from a liquid solution of ions. Any pair of complementary ions that come into proximity bond together to form a precipitating molecule, removing the ion pair from the solution.

Equality

We take fusion-with-an-inverse as defining equality:

a=a iff a|Δa = ><
a=b iff a|Δb = ><

Conventionally b is a different structure than a, with an equivalent value. Structural equivalence is identity,

a=a iff a|Δa = ><

Thus another form of equality as fusion is

a=b iff a|Δb = a|Δa

===
collect other pieces about =
===

Forced Type Matching

When an in-from contains no instances of the for-form, the substitution fails to match. This is usually interpreted as no change to the in-form. For application to unit ensembles, we desire an interpretation for which a match is always guaranteed, since wholes are composed of units and only units. Units can

fail to match only when the type of unit that constitutes the for-form differs from the type of unit that constitutes the in-form. Thus for example, the for-form does not match the in-form in the substitution

[●● ◊ ●●●]

We can, however, apply changeUnit to assure that a match occurs. In the example,

[●● ◊ ●●●] = Δ[●● ◊ Δ●●●]
= Δ[●● ◊ ◊◊◊]
= Δ●●●●●●
= ◊◊◊◊◊◊

Forced type matching permits substitution to be interpretable regardless of unit types. Forced type matching is defined as,

when (●c and ◊e) or (◊c and ●e),
[a c e] ==> Δ[a c Δe]

Since any whole consists of identical units, we can consider only the base case of each unit without loss of generality. The four base cases which require forced type matching are:

[● ● ◊] = Δ[● ● Δ◊] = Δ[● ● ●] = Δ● = ◊
[● ◊ ●] = Δ[● ◊ Δ●] = Δ[● ◊ ◊] = Δ● = ◊
[◊ ● ◊] = Δ[◊ ● Δ◊] = Δ[◊ ● ●] = Δ◊ = ●
[◊ ◊ ●] = Δ[◊ ◊ Δ●] = Δ[◊ ◊ ◊] = Δ◊ = ●

In any substitution, the type of the put-form does not effect identification of a match, so that we only need consider the first two of the above cases, for which the for-forms and the in-forms differ in unit type.

The result of the substitution in all four cases is changeUnit applied to the put-form. In general, the consequence of any application of forced type matching will be changeUnit applied to the substitution result. That is to say, forced type matching is equivalent to changing the unit of the result of the substitution. We will embed this result into the definition of substitution:

[a c e] =def= substitute a for c in e, ignoring the type of c and e;
should the type of c and e not match, apply changeUnit to the result

In other words,

when (●c and ●e) or (◊c and ◊e), [a c e]

when (●c and ◊e) or (◊c and ●e), Δ[a c Δe]

Note that forced type matching does not imply forced matching. Even when the type of the for-form and the in-form match, there are cases for which no match will occur. For example:

[● ●● ●]

In this case, there cannot be a for-form match within the smaller in-form. In such cases, we will still not require that the failed match be equal to the unchanged in-form. Instead, the failed substitution will simply not be enacted, and thus will not be reduced. This is of advantage in cases where the context of the failed substitution may include an interaction with other forms that may later permit the substitution to occur.

Absolute Typing

Analogous to absolute value for integers, absolute typing treats the type of all units as the same. Absolute typing, that is absence of typing, is an alternative to forced-type-matching that can clarify what it means to "ignore the type of a form". Type-blind substitution treats all units as (nominally) Solid, assuring that forced type matching is never invoked.

Thus an alternative definition of substitution over different types would be to convert all for-forms and in-forms to Solid type, apply substitution, and then apply changeUnit to the result in cases that the types of the for-form and the in-forms do not match.

In the interpretation for integers, this approach is equivalent to the sign rule for multiplication.

Distribution of ChangeUnit over Substitution

ChangeUnit can also be applied across substitutions. Two applications of changeUnit maintain the value of the substitution, regardless of which substitution form they are applied to:

[a c e] = Δ[Δa c e] = Δ[a Δc e] = Δ[a c Δe]

As well,

[a c e] = [Δa Δc e] = [a Δc Δe] = [Δa c Δe]

Two applications of changeUnit do not change the result of a substitution, regardless of which forms they are applied to. We separate the possible substitution varieties into those that require forced type matching and those that do not, applying changeUnit when necessary to achieve forced type matching.

Without forced type matching, there are four possible base cases:

[● ● ●] = ●
[◊ ● ●] = ◊
[● ◊ ◊] = ●
[◊ ◊ ◊] = ◊

We consider all of them abstractly as [a c e]. Applying changeUnit twice results in six different patterns:

Δ[Δa c e]
Δ[a Δc e]
Δ[a c Δe]
[Δa c Δe]
[Δa Δc e]
[a Δc Δe]

Δ[Δa c e]: In all cases, the result of the substitution will be the put-form. Should the put-form a be changed to Δa, the result will be Δa. Applying the outer changeUnit returns the result to a, since

ΔΔa = a

[a Δc Δe]: In all cases, the result of the substitution will be the put-form. Changing units in c and e will not effect the result since the same substitution will occur.

Δ[a Δc e], Δ[a c Δe], [Δa c Δe], and [Δa Δc e]: For each of these cases, forced type matching will be triggered:

Δ[a Δc e] ==> ΔΔ[a Δc Δe] = [a Δc Δe]
Δ[a c Δe] ==> ΔΔ[a c ΔΔe] = [a c e]
[Δa c Δe] ==> Δ[Δa c ΔΔe] = Δ[Δa c e]
[Δa Δc e] ==> Δ[Δa Δc Δe]

The last of these results can be decomposed into single applications of the first and second results above.

The four cases that require forced type matching are shown in the previous sub-section. By applying forced type matching, each converts uniquely into a

natural substitution for which the types for the for-form and the in-form match. The result of a natural substitution will be the type of the put-form. The final application of changeUnit toggles the type of the result.

A Simpler ChangeUnit

We have defined changeUnit as

$$\Delta E \ =def= \ [\bullet/\diamond \ \diamond/\bullet \ E]$$

With substitution-with-forced-type-matching, we can say

$$\Delta E =def= [\bullet \ \diamond \ E] \quad \text{with} \quad [\bullet \ \diamond \ E] = [\diamond \ \bullet \ E]$$

The application of changeUnit changes all units in a given ensemble:

$$\Delta_{\bullet}E = {}_{\diamond}E$$
$$\Delta_{\diamond}E = {}_{\bullet}E$$

Substitution with forced-type-matching compares the type of the for-form to the type of the in-form. When they differ, the type of E is changed.

when ${}_{\bullet}E$, $[\bullet \ \diamond \ E] \Longrightarrow [\bullet \ \bullet \ \Delta E] = \Delta E$
$[\diamond \ \bullet \ E] = {}_{\diamond}E = \Delta E$
when ${}_{\diamond}E$, $[\bullet \ \diamond \ E] = {}_{\bullet}E = \Delta E$
$[\diamond \ \bullet \ E] \Longrightarrow [\diamond \ \bullet \ \Delta E] = \Delta E$

This permits the simpler definition of changeUnit since forced-type-matching manages difference between for-form and in-form types.

Fusion Composition/Decomposition via ChangeUnit

Applying function substitution to fusion yields:

if a=b then a|n = b|n

With changeUnit, Fusion Decomposition can be proved directly.

Assume a=b

$a|\Delta b = ><$
$a|\Delta b|n|\Delta n = ><$
$a|n|\Delta b|\Delta n = ><$

a|n|Δ.b|n = ><

a|n = b|n

When equivalent Wholes are each fused with the same Whole, equivalency is maintained. This provides a composition rule for fusions. The above algebraic proof reads in both directions, providing both composition and decomposition rules for fusion. In all cases,

a=b iff a|n = b|n

```
================================================================================
COMPLETE AXIOMATIZATION of UNIT ENSEMBLES with Annihilation

      S(●)                                                  sort
      H(◊)                                                  sort
      ●a  =def=  S(a)                                       type-of notation
      ◊a  =def=  H(a)                                       type-of notation
      W(a)  iff  S(a) xor H(a)                              sort

      A = A                                                 equals identity
      A = B  iff  B = A                                     equals symmetry
      A = B  and  B = C  iff  A = C                         equals transitivity

      fuse(a|...|z)  =def=  a...z                           fusion definition
      fuse(a) = a                                           whole fusion
      W(fuse(a|b))  iff  W(a) and W(b)                      uniqueness
      a = ●  xor  a = ◊  xor  a = fuse(b|c)                 no void wholes
      fuse(●|◊) = ● ◊  =  «void»                            annihilation
      a=b  iff  fuse(a|c) = fuse(b|c)                       composition/decomposition

      sub(«void»,C,E) = delete(C,E)                         put-void substitution
      sub(A,«void»,E) = construct(A,E)                      for-void substitution
      sub(A,C,«void») = «void»                              in-void substitution
      sub(A,E,E) = A                                        global substitution
      sub(A,A,E) = E                                        self substitution

      if A=B then  sub(A,C,E) = sub(B,C,E)                  put-equality
      if C=D then  sub(A,C,E) = sub(A,D,E)                  for-equality
      if E=F then  sub(A,C,E) = sub(A,C,F)                  in-equality

      sub(A,C,E) = E  iff  A = C                            value maintenance
      sub(A,C,E=F)  =  sub(A,C,E)=sub(A,C,F)                distribution

      sub(fuse(a|b), C, fuse(e|f)) =
            fuse(sub(a,C,e)|sub(b,C,e)|sub(a,C,f)|sub(b,C,f))     distribution

      changeUnit(E) = ΔE  =def=  [● ◊ E] = [◊ ● E]          changeUnit definition

      Δ● = ◊                                                base case changeUnit
      Δ◊ = ●                                                base case changeUnit
      fuse(a|Δa) = «void»                                   identity changeUnit
      Δfuse(a|b) = fuse(Δa|Δb)                              distribution
      a=b  iff  a|Δb = «void»                               equality
================================================================================
```

COMPLETE AXIOMATIZATION of UNIT ENSEMBLES with Annihilation (condensed notation)

S(●)	sort
H(◊)	sort
●a =def= S(a)	type-of notation
◊a =def= H(a)	type-of notation
W(a) iff ●a xor ◊a	sort
A = A	equals identity
A = B iff B = A	equals symmetry
A = B and B = C iff A = C	equals transitivity
a|...|z =def= a...z	fusion definition
(a) = a	whole fusion
W(a|b) iff W(a) and W(b)	uniqueness
a = ● xor a = ◊ xor a = b|c	no void wholes
a=b iff a|c = b|c	composition/decomposition
●|◊ = ● ◊ = ><	annihilation
[>< C E] = delete(C,E)	put-void substitution
[A >< E] = construct(A,E)	for-void substitution
[A C ><] = ><	in-void substitution
[A E E] = A	global substitution
[A A E] = E	self substitution
[A C E] = [B C E] iff A = B	put-equality
[A C E] = [A D E] iff C = D	for-equality
[A C E] = [A C F] iff E = F	in-equality
[A C E] = E iff A = C	value maintenance
[A C E=F] = [A C E]=[A C F]	distribution
[a|b c e|f] = [a c e]|[b c e]|[a c f]|[b c f]	distribution
ΔE =def= [● ◊ E]	changeUnit definition
Δ● = ◊	base case changeUnit
Δ◊ = ●	base case changeUnit
a|Δa = ><	identity changeUnit
Δ.a|b = Δa|Δb	distribution
a=b iff a|Δb = ><	equality

```
================================================================================
AXIOMS of UNIT ENSEMBLES with Annihilation

      S(●)                                                    sort
      H(◊)                                                    sort
      W(a)  iff  S(a) xor H(a)                                sort

      a|...|z  =def=  a...z                                   fusion definition
      W(a|b)  iff  W(a) and W(b)                              uniqueness
      ●|◊ = ● ◊  =  ><                                        annihilation

==============
GIVEN EQUALITY
      A = A                                                   equals identity
      A = B  iff  B = A                                       equals symmetry
      A = B  and  B = C  iff  A = C                           equals transitivity
      [A C E] = [B C E]  iff  A = B                           put-equality
      [A C E] = [A D E]  iff  C = D                           for-equality
      [A C E] = [A C F]  iff  E = F                           in-equality
      [A C E] = E  iff  A = C                                 value maintenance
      [A C E=F]  =  [A C E]=[A C F]                           distribution
      [a|b c e|f] = [a c e]|[b c e]|[a c f]|[b c f]            distribution
Consequences
      (a) = a                                                 whole fusion
      Δ● = ◊                                                  base case changeUnit
      Δ◊ = ●                                                  base case changeUnit
      [A C ><] = ><                                           in-void substitution
      [A E E] = A                                             global substitution
      [A A E] = E                                             self substitution
Provable?
      a = ●  xor  a = ◊  xor  a = b|c                         no void wholes
      a=b  iff  a|c = b|c                                     composition/decomposition
      a|Δa = ><                                               identity changeUnit
      Δ.a|b = Δa|Δb                                           distribution
      a=b  iff  a|Δb = ><                                     equality
Definition
      [>< C E] = delete(C,E)                                  put-void substitution
      [A >< E] = construct(A,E)                               for-void substitution
      ΔE  =def=  [● ◊ E]                                      changeUnit definition
================================================================================
```

UNIT AND BOUNDARY ARITHMETICS -- SECTION IVA October 1, 2008 10:55 PM
William Bricken
January 2007

UNIT ENSEMBLE ARITHMETIC
Theories
Group Theory
Successors and Induction

===

The Structure of Domain Theories

A domain theory (or abstract knowledge structure) consists of a domain of objects, and axioms and rules which define the symbolic interaction between the symbolic form of these objects. In particular, a domain theory consists of:

1. A collection of symbols, including
   constants
   variables naming arbitrary forms
   functions
   relations

2. Generation axioms
   These define the typing hierarchy of forms

3. Uniqueness axioms
   These define how forms stay the same when they are manipulated, and how forms are composed of atomic units.

4. Special axioms
   These define the characteristics of special types.

5. An Induction Principle
   This rule template is the mechanism which allows construction and deconstruction of arbitrary forms, and provides an algebraic (abstract) approach to domain forms.

For proof and for programming, several composition tools are then proved/provided for construction and deconstruction.

6. Decomposition
   Permission to take apart an arbitrary form into atomic components and functions to do the construction/deconstruction.

7. Equality under Decomposition
   Equal forms don't change if you do equivalent things to them. Generally, forms are mappable, you can map a function across the atomic parts.

8. Special functions as theorems
   With the above basis (1-7), we now begin to build specialized functions (macros) which make it easier to take large steps while manipulating forms. A recursive definition axiom says what we mean by the new function in terms of the basis functions. Then other theorems relate all the other mechanisms to the

new function. Generally each new function has analogous axioms for each item above.

===

Domain Theories

A domain is the collection of simple objects which are of (mathematical) interest. Generally the labels of objects in a domain refer, or point, to concrete objects in reality. A domain theory consists of a collection of objects, together with a particular set of functions and relations which define and constrain the generic behavior of both simple and complex objects in the domain.

Domain theories have a specific mathematical form which not only identifies how the objects in that domain behave, but also provides all the information needed to write processing algorithms for the domain objects. The prototypical components of a domain theory are

- a representation of the elementary unit or constants, the base of the structure (also called the carrier set)
- recognizer predicates which identify the particular types of structure
- a constructor function which builds compound structures from simple units
- an accessor function which gets parts of a compound structure
- a collection of functions which transfer between domain objects
- a collection of invariants, or equations, which define the structure's behavior
- an induction principle which specifies how to verify correct manipulations

In an algebraic theory, you usually also have variables, names which are generic, standing in place of an arbitrary member of the domain base.

Domains with Internal Structure

To add descriptive complexity, we add internal structure to propositions. There are two general classes of structure: relations and functions. Relations are connections, or structures, holding together pairs of simple objects. Functions are a restricted type of relation, one that permits functions to stand in place of object names. Functions are relations which name objects in a domain by using other object names.

In computer science, we refer to complex objects as data structures, and the set of relational constraints on these objects as abstract data types.

The important idea is that all data structures, all domains, have the same organizational structure. All domains and data types consist of a collection of these axiomatic principles:

Labels
Recognizers
Constructors
Accessors
Functions
Invariants (relations)
Induction Principle

In object-oriented approaches, the abstract data type includes all algorithmic functionality. That is, using oo-techniques, the above principles define the entirety of an object, and thus the entirety of a program.

Three examples of domain theories follow. These examples are schematic outlines, intended to suggest both mathematical approach and coding technique. Each domain has additional functions and relations which are not included here.

The Domain Theory of Non-negative Integers

Base	`0`
Objects	`{positive integers}`
Recognizer	`integer[n]`
Constructor	`+1[n], inc[n]`
Accessor	`-1[n], dec[n]`
Decomposition axioms	`(integer[n] and not[n=0]) -> (+1[-1[n]] = n)` `integer[n] -> (-1[+1[n]] = n)`
Uniqueness axiom	`(+1[n1] = +1[n2]) iff n1=n2`

Functions	`+:` (associative, commutative, identity=0)
	`n+0 = n`
	`n1 + +1[n2] = +1[n1+n2]`
	`(n1=n2) -> (n1+n3) = (n2+n3)`
	
	`-:`
	`n-0 = n`
	`+1[n1] - +1[n2] = n1-n2`
	
	`*:` (associative, commutative, identity=1)
	`n*0 = 0`
	`n1*(n2+1) = n1*n2 + n1`
	
	`^:`
	`n^0 = 1`

```
n1^(n2+1) = (n1^n2)*n1
```

Some invariants

```
integer[n] or not[integer[n]]
integer[+1[n]]
integer[0]
not[+1[n] = 0]
integer[n1+n2]
+1[0] = 1
n+1 = +1[n]
n^1 = n
if not[n=0] then 0^n = 0
```

Induction

```
if (F[0] and (F[n] -> F[+1[n]])) then F[n]
```

Decomposition Induction

```
if (F[0] and (F[-1[n]] -> F[n]) then F[n]
```

The Domain Theory of Sets

Set Base

```
{} Phi (the empty set)
```

Element Base

```
{a,b,c,...} from some domain
```

Objects

```
{S1,S2,...}     Universe = PowerSet[elements]
```

Recognizers

```
atom[a]           set[S]
```

Constructor

```
a•S, insert atom a into set S
```

Accessor

```
member[a,S]
member[choice[S],S]
not[member[choice[S],rest[S]]]
```

Decomposition axiom

```
(not[S=Phi]) -> (S = (choice[S] • rest[S]))
```

Uniqueness axiom

```
(member[a,b•S] iff (a=b) or member[a,S]
```

Functions

Equality:

```
S1=S2 iff (choice[S1]=choice[S2] and
  member[choice[S1]] and member[choice[S2]])
```

Intersection: (associative, commutative, idempotent, identity=Universe)

```
intersect[Phi,S] = Phi
intersect[a•S1,S2] = if member[a,S2]
          then (a•intersect[S1,S2]) else intersect[S1,S2]
```

Symmetric-difference: (associative, commutative, identity=Phi)

```
sym-diff[S1,Phi] = Phi
sym-diff[S1,S2] = if (member[a,S1] and not[member[a,S2]])
            or (member[a,S2] and not[member[a,S1]])
```

```
                    then member[a,sym-diff[S1,S2]]

    Cardinality, #:
            #[Phi] = 0
            if not[member[a,S]] then #[a•S] = #[S] + 1
```

Some invariants

```
set[Phi]
set[a•S]
not[member[a,Phi]]
(intersect[{a},{b}] = Phi) -> (not[a=b])
intersect[S,Phi] = Phi
member[a,intersect[S1,S2]] iff
    member[a,S1] and member[a,S2]
S1 intersect (S2 sym-diff S3) =
    (S1 intersect S2) sym-diff (s1 intersect S3)
```

Induction

```
if F[Phi] and if not[member[a,S]] then (F[a] -> F[a•S])
   then F[S]
```

===

The foundations of arithmetic define mathematical objects in terms of set theory and predicate calculus. A mathematical system consists of a set of objects and specific relations and functions associated with these objects. Relations and functions are defined in terms of mappings between a set of objects called the domain and a set of objects called the range. The axioms and definitions that define the behavior of a mathematical system are called a theory. Both proof and computation rely heavily on the Induction Principle.

In this section we will consider several conventional theories that incorporate sets of integers and functions and relations over the integers. They include comparison, addition, multiplication, subtraction, division, exponents, and fractions. We will show the mappings between the system of unit ensembles and the conventional axioms of integer arithmetic.

The objective is to demonstrate that unit ensembles fully cover the well-known theories of rational arithmetic, and further that unit ensembles cover rational arithmetic elegantly. Elegance means fewer and more intuitive concepts and more efficient mechanisms for computation.

The theory of unit ensembles is an algebraic theory with

equality as the primary relation
ensembles of identical units as models of integers
two types of units (Solid and Hollow)
one comparison predicate (greater-than)
one variary operation (fusion)
one ternary operation (substitution), and
one utility function (changeunit).

Axioms provide existence and uniqueness theorems for constructing ensembles, and are generally expressed algebraically. Computation proceeds by algebraic substitution, following a collection of equality rules, and one special void-based rule, annihilation. Proof is algebraic and incorporates no additional principles.

In comparison, the theories of rational arithmetic are also algebraic, with

equality as the primary relation
one set of unique objects (integers) with one type of object (integer)
one set of object pairs (fractions)
one comparison predicate (usually weak less-than)
one unary operator (negative)

five binary operators (add, subtract, multiply, divide, power)
several utility functions (max, min, ...%%%), and
Predicate Calculus and Set Theory as semantic context.

Axioms provide existence and uniqueness theorems for objects joined by operators, and are expressed in Predicate Calculus. Computation proceeds following collections of both inferential and equational rules. Proof incorporates the rules of inference and the Induction Principle.

We can see that these two representational structures are quite simiular. Their substantive differences include:

Structure	Unit Ensemble Arithmetic	Set Theory
Integers	identical unit replicates	infinite classes of sets with the same cardinality
Signs	two discrete unit types	one type and one operator
Operations	fusion and substitution	+,-,*,÷,^
Tools	algebraic substitution	substitution, induction, inference
Concepts	void, flatness, concurrent	associative, commutative, zero, inverses

The essential pragmatic differences:

spatial, visual, kinesthetic, intuitive	abstract
parallel	sequential
two operations	five operations

Group Theory

From the perspective of group theory, systems of integers are commutative rings (associative, commutative, distributive) with zeros and inverses. Systems of unit ensembles do not fit conventional group theoretic structures. Ensembles are not set theoretic objects. Although ensembles are unique, their parts are not. There are two types of units that mutually annihilate. There is no additive zero, and addition is pre-associative and pre-commutative. Addition and subtraction are differentiated by unit types rather than by inverse operations. Multiplication and division are also joined into one operation,

that of substitution. Multiplication-as-substitution unites the concept of multiplication with algebraic substitution rules provided by equality. This leaves only one multiplication axiom, that of commutativity of substitutions. Figure %%% shows the group theoretic description of arithmetic, while Figure %%% shows the axioms of unit ensemble arithmetic.

===

Group Theory

Algebraic systems ((S,f), where S is a set and f is a binary function on that set) can be classified into groups having similar structural characteristics. This additional level of abstraction is called group theory, or modern algebra.

The essential distinguishing characteristics of algebraic systems (S,f):

Let a,b,c inS and e, the identity element, inS

Closed binary operation: f(a,b) = c

Associativity: f(f(a,b),c) = f(a,f(b,c))

Identity element: Exists e inS. f(e,a) = f(a,e) = a

Inverse element: Exists y inS. f(a,y) = f(y,a) = e

Commutativity: f(a,b) = f(b,a)

Types of Algebraic Systems

Groupoid: (S,f) such that S =/= { }

Loop: Groupoid and
All a,b,c in S. if f(a,b) = f(a,c) then b=c
if f(a,c) = f(b,c) then a=b

Semigroup: Groupoid and
S is closed under f
f is associative on S

Monoid: Semigroup and
(S,f) has an identity element

Group: Monoid and
every element in S has an inverse.

Each type can be combined with the commutative property, to give

commutative loop
commutative groupoid
commutative semigroup
commutative monoid
commutative group (boolean algebra is an example in this category)

===

additive

inverse	a + (-a) = 0	minus a is a new object
identity	no operation	
commutative	not representative, not relevant	
associative	not representative, not relevant	

multiplicative

inverse	a * (1/a) = 1	reciprocal of a is a new object
identity	a * 1 = a	
commutative	a * b = b * a	
associative	a * (b * c) = (a * b) * c	

distributive a * (b + c) = a*b + a*c

===

Successors and Induction

Doing without Relations, Mathematical Induction and Iteration

a > b does not require a difference, just the sign of the difference:

a − b > 0 or Positive[a−b]

a = b can be tested by

a − b = 0

Each • in b gets ◊, apply • ◊ = void

• is a number
If a and b are numbers, so is a b (sharing the same space)

Every number has many names, but only one cardinality.

Peano

N0

==========

M. Kline (1980) Mathematics: the Loss of Certainty. Oxford

R. Descartes (1641) Meditations

P. Benacerraf and H. Putnam (eds) (1983) Philosophy of Mathematics (second edition) Cambridge
===

===
Proof

Boolean algebra axioms and theorems (valid transformations) provide a way to explore decision spaces without making the actual decisions. This is called logical or algebraic proof. However, the situation remains complex because now we must select which theorem to apply and where to apply it. Although the search space is more abstract, it is still intractable and inconvenient. Although Boolean algebra abstracts the physical properties of decisions, it is still a real world problem to use Boolean algebra efficiently.

Mathematical:

```
if (base-case is true) then base-value else F[recursive-step]
```

Mathematical Induction

Induction depends on a order relation over a domain U. The idea is to demonstrate truth for the base case (the simplest member of the ordered set), and then to demonstrate the truth for an arbitrary member of the set, assuming the truth of the member next to it in the order relation.

```
If N is an ordered set and property P isTrue for
      1) the minimal member of N, and
      2) if P(x) then P(next(x))
then P isTrue for all members x of N.
```

Using the natural numbers, N = {1, 2,...}:

```
If P(1) isTrue, and
      assuming P(x) we can show that P(x+1) isTrue, then
P(x) isTrue for all members of N.
```

Some Inductive Definitions

Base case: the value of the most elementary case

Examples:

`zero`	the additive identity
`one`	the multiplicative identity
`Phi`	the empty set
`nil`	the empty list, the empty tree
`false`	the logical ground

Generating rule: the transform which defines the next case, given an arbitrary case

Examples:

```
successor[n]      =     current[n] + 1
power-of-2[n]     =     2 * current[n]
summation[n]      =     n + current[n]
last[list]        =     rest[list] = nil
length[list]      =     length[rest[list]] + 1
member[x,S]       =     x=select[S] or member[x,rest[S]]
power-set[S]      =     current[S] * S
cardinality[S]    =     cardinality[rest[S]] + 1
node[btree]       =     left[btree] + right[btree]
logic-form[lf]    =     current[lf] implies next[lf]
parenthesis[pf]   =     "(" + current[pf] + ")" or
                              current[pf] + next[pf]
```

An inductive definition consists of three components:

- a base case, the simplest possible application of the induction
- an inductive case which assumes an arbitrary member of the domain, and constructs the adjacent member.
- an ordering principle which provides a structure for inferring that when one member can be constructed from adjacent member, then all members can be constructed.

Induction Principles

When the three components of an inductive definition are combined, they produce an Induction Principle for the particular domain. Inductive principles are second order functions, or functionals. We are most familiar with first-order functions, which vary over domain objects (ie variables). Second-order functions vary over other functions; the domain is a set of functions rather than a set of objects. Sometimes second-order functions are called functional schema, they are patterns, or schema, which specify relations between objects which hold for a set of functions. Each schema can be instantiated to many (often infinite) specific induction rules for specific functions.

Generic Induction Schema

Primitive recursive schema without parameters for the integer domain:

```
f[n] =def=
            if n=0 then k else h[n-1,f[n-1]]
```

A function acting on an integer argument `n` can be generically thought of as follows:

If `n` is `0`, return some constant `k`,
otherwise apply function `h` with the arguments one step closer to `0`
[that is, with arguments `(n-1)` and `f` applied to `(n-1)`].

Example: the factorial function

```
f = fac     k = 1       h = *       h[p,q] = (p+1)*q

fac[n] =def=
            if n=0 then 1 else n*fac[n-1]
```

Primitive recursive schema with parameters for the integer domain
(for simplicity, only one parameter is shown here):

```
f[n,m] =def=
            if n=k then g[m] else h[n-1,m,f[n-1,m]]
```

Example: the times function

```
f = *       k = 1       g[m] = m    h = +       h[p,q,r] = q*(r+1)

*[n,m] =def=
            if n=1 then m else +[m,*[n-1,m]]
```

Predicate Logic

Notation: truth symbols = {T,F}
constant symbols = {a,b,...}
variable symbols = {x,y,...}
function sysmbols = {f,g,...}
relation symbols = {p,q,...}

Terms: Constants and variables are terms.
If {t1,t2,...} are terms, and f has arity n, then f[t1,...,tn] is a term.

Relations: Truth symbols are relations.
If {t1,t2,...} are terms, and p has arity n, then p[t1,...,tn] is a relation.

Sentences: Terms are sentences.
Relations are sentences.
If S1 and S2 are sentences, then so is S1->S2.
If S is a sentence, then so is (all x.S)

Integers

Notation:

```
Successor[n] = n'

integer[1]
if integer[x] and integer[y] then integer[x+y]
```

Counting:

```
not[x'=0]
1      = 0'
n + 1  = n'
```

Addition:

```
m + 0  = m
m + n' = (m + n)'
```

Multiplication:

```
m * 0  = 0
m * n' = (m * n) + n
```

Exponentiation:

```
m ^ 0  = 1
m ^ n' = (m ^ n) * n
```

```
sum[0] = 0
sum[i'] = sum[i] + i'

fac[0] = 1
fac[i'] = fac[i] * i'

sumfac[0] = 1
```

```
sumfac[i'] = fac[i'] + sumfac[i]

fib[1] = fib[2] = 1
fib[i''] = fib[i'] + fib[i]

power-of-2[0] = 1
power-of-2[n'] = 2*power-of-2[n]

0 = 0
m' = n' iff m = n
```

Sets

`{}` is a set.
If `S` is a set, then so is `u•S`

```
powerset[{}] = {{}}
powerset[S+{e}] = powerset[S]*(S+e)

member[u,{}] = F
member[u,v•S] iff u=v or member[u,S]

add-to-set[u,{}] = {u}
add-to-set[u,v•S] = if u=v then S else add-to-set[u,S]

{} = {}
u•S1 = v•S2 iff u=v and S1=S2


union[{},S] = S
union[u•S1,S2] = if member[u,S2] then union[S1,S2]
                         else u•union[S1,S2]

intersection[{},S] = {}
intersection[S,{}] = {}
intersection[u•S1,S2] = if member[u,S2] then u•intersection[S1,S2]
                                else intersection[S1,S2]

subset[{},S2] = T
subset[S1,{}] = F
subset[u•S1,S2] = member[u,S2] and subset[S1,S2]

proper-subset[S1,S2] iff subset[S1,S2] and not[S1=S2]

cardinality[{}] = 0
cardinality[u•S] = if [member[u,S]] then cardinality[S] + 1
                                else cardinality[S]
```

===
Proof Needs Semantics

This example illustrates why automated proof (and intelligent computation) is very unlikely. We must design programming and verification systems to be interactive, so that they augment human intelligence rather than attempting to emulate it. Note that conventional programming puts the interaction in a batch mode.

```
For any natural number n>=2, (n^3-n)/6 is an Integer

      <[n,2] or isInt[div[minus[power[n,3],n],6]]

      L[n,2] or P[Q[R[S[n,3],n],6]]
```

where the semantics is

```
L is <            less than
P is  isInt       type check, to be proved
Q is  /           divide
R is  -           subtract
S is  ^           power
```

A person may elect to factor the expression in question, in order to understand more:

```
(n^3-n) = n*(n^2-1) = n*(n+1)*(n-1) = (n+1)*n*(n-1)
```

A machine can do this, not in order to understand, but as part of a set of automated transformations to be explored.

```
R[S[n,x],n]  =>  T[n,R[S[n,R[x,1]],1]]

             =>  T[n,T[U[n,1],S[n,1]]]
```

where

```
T is  *           multiply
U is  +           add
```

Note how very specific these patterns are. These rules could be generalized, but it is difficult to know in advance in which direction the generalization should be formulated.

Number Facts

A human might next retrieve a collection of esoteric number facts:

> For any three numbers in a row,
> there must be at least one even number
> and at least one number divisible by three
> and these two numbers are not the same number.
>
> so, the product contains divisors of 2 and 3 (i.e. 6)

A machine can't make this step, because there are too many esoteric number facts. Even with very sophisticated meta-knowledge to steer the selection of which number facts to explore first, finding

the correct set of facts which leads to a proof is generally not possible. The problem, for example, may be only slightly different, but would then require entirely different esoteric number facts:

```
if n is odd, then (n^3-n)/8 is an Integer

      div[n,2]
            or
      isInt[div[minus[power[n,3],n],8]]

      Q[n,2] or P[Q[R[S[n,3],n],8]]
```

Induction

We simply do not know if there are automated paths, using different proof strategies, which reduce all mathematical problems to trivialities. For example, machines can do induction. Rather than recalling esoteric number facts to generate a natural, intelligent proof, we could have gone blindly forth in the above problem, trying an inductive proof:

```
      For any natural number n>=2, (n^3-n)/6 is an Integer
```

[[n • n • n]|Δn 6 •] = [n • n • n 6 •]|[Δn 6 •]

= [n • n 6 n]|[Δn 6 •]

show [[k|• • k|• • k|•]|Δ.k|• 6 •]

o

base:	`(2^3-2)/6 isInt`	(the base case is not n=0)
general:	`(k^3-k)/6 isInt`	
show:	`((k+1)^3 - (k+1))/6 isInt`	

```
      ((k+1)^3 - (k+1))/6

      = (k^3 +3k^2 + 3k +1 - k - 1)/6

      = (k^3-k)/6 + 3(k^2+k)/6

if A isInt and B isInt, then (A + B) isInt
```

`(k^3-k)/6 isInt`		assume general
`3(k^2+k)/6 isInt`		to show
`k(k+1)/2 isInt`		lemma

```
((k+1)^3 - (k+1))/6     isInt       QED
```

Lemma: `k(k+1)/2  isInt`

base: `2(2+1)/2  isInt`

general: `k(k+1)/2  isInt`

show: `(k+1)(k+1+1)/2  isInt`

```
(k+1)(k+2)/2
= (k^2 + 3k + 2)/2
= k(k+1)/2 + 2(k+1)/2
```

`k(k+1)/2  isInt` assume general

`2(k+1)/2  isInt` k isInt

`(k+1)(k+2)/2  isInt` QED

Existence Proof

An existence proof demonstrates that a particular object, or solution, does exist, but the proof does not identify exactly what that object is. Automated systems cannot conduct existence proofs.

Essentially, existence proofs demonstrate universal principles, whereas computational proofs demonstrate a verification of a particular principle.

Prove there exists a function that is both odd and even.

Odd function: `F[-x] = -F[x]` e.g. sine

Even function: `F[-x] = F[x]` e.g. cosine

Find an `F[x]` such that:

`E[F[R[0,x]], R[0,F[x]]]` and `E[F[R[0,x]],F[x]]`

where `E is =` and `R is -`

An example of such a function is: `F[x] = 0`

For pattern-matching, we are looking for two different matches to the second argument of `E`. Equivalently, we can eliminate `F[-x]` algebraically, so that we are looking for the single pattern:

```
F[x] = -F[x]

E[F[x], R[0,F[x]]]
```

From here, the example of `F[x]=0` is easy to identify, since `R[0,0]=0` and `E[0,0] isTrue`. In general, the problem is to show:

```
Exists F.   E[F[x],R[0,F[x]]]
```

One problem is that the existential quantifier is over *functions*, not variables. This requires Second Order Logic, for which theorem provers are not yet well developed. What is the Domain of all Functions? How do we enumerate, even recursively, all possible functions?

===

Peano's axiomatization of whole numbers is based on incremental units (the successor, or +1, function), defining whole numbers as a sequence of nested successor functions, similar to counting in unit increments. Proofs call upon mathematical induction over successor strings. Mathematic induction is a technique for assuring the correctness of a mathematic argument. Its simplest form utilizes the succession of integers from 1 to say N. Induction an Peano's successor function are intimately connected.

To use induction two cases are required, a ground, or base case, and an inductive case. The inductive ground is the explicit zero of our number system. When we make a mathematical assertion based on induction, we first show that the assertion is true for zero. The inductive step requires choosing an arbitrary number, called N, to assume the assertion true for. Then, through algebraic and logical argument, the assertion is shown to be true for the next number N+1, also called s(N) for "successor function applied to N". If we assume that the assertion is true for N, and we can show that it also true for N+1, then induction is simply the idea of taking that incremental step again to N+2, and again to N+3, and continuing until as far as desired. Since we began showing the assertion true for zero, and have selected an arbitrary N that could be any number, then in sum we have shown that the assertion is true for all numbers. Sort of: If I can get started at 0 and then get to 1, I'll do the same thing to get to 2, and so on.

For an example of conventional induction, we will show that the sum of any two positive numbers is always greater than or equal to one of those numbers. Yes, this seems obvious, but the program of mathematical formalism is to eliminate what "seems" in favor of what can be rigorously proven.

```
x + y ≥ x     ? x+z=y iff x≤y       x≤x+z
```

```
                              x=0, 0≤0+z  let z=1
                              x<0, no x
                              x>0, induct y is arbitrary
                                    assume x+z=y-1 iff x<y-1
                                    show x≤y iff x+z'=y
      know  x=y  or  x≤y-1 from def of ≤
            case x=y          x + z' = x    yes when z'=0
            case x≤y-1        assume x+z=y-1, x+z+1=y-1+1
                                    => x+z+1=y, let z'=z+1 in x+z'=y
```

The example induction will be on the number x, that is, we will vary x from 0 to N. First the base case of zero:

$$0 + y \geq 0$$

Yes, this statement is true, since zero adds nothing to y, and the definition of the comparison relation, ≥, is consistent with the assumption that y is a positive number.

$$y \geq 0 \qquad \text{known}$$

At this point we know that y is a legitimate positive number. Now we assume that the assertion is true for N,

$$N + y \geq N \qquad \text{assumed}$$

And from there, we need to show that the assertion is true for N+1:

$$(N+1) + y' \geq (N+1) \qquad ?$$

$$N + 1 + y' \geq N + 1 \qquad ?$$

===junkV?
The natural tendency is now to "subtract one to each side" of the comparison. But in the realm of foundations, we can use only what is known and assumed. Here, subtract-one is not permitted because we have defined a successor function, not a predecessor function. Why not just define a predecessor function? Well that is properly pragmatic, but the particular foundational game here is to use the absolute minimum of tools and structure.

We can not, for example, apply add-one to each N-1, since this would be returning to the assumption, N, directly, without having addressed what is being defined. Here, the concept of "adding one to any number" is the inverse of the concept "subtracting one from any number". There is also an analogous

application of induction starting at N and descending to N-1, and on and on until 0.
===^

Induction treats positional notation strictly as a notational convenience. Peano arithmetic is, however, strictly linear; although proof can bypass counting through the use of the Induction Principle, computation must iterate through each number from 1 to N in unit increments. The formal axiomatization of arithmetic incorporates induction as a technique to render unit arithmetic tractable for proof, but pragmatic usage incorporates place-value notation as a technique to render arithmetic tractable for computation.

===
PROOF OF THE LEAST NUMBER PRINCIPLE WITHOUT INDUCTION
William Bricken
December 2006

equivalent to the complete induction principle

if forsome-integer x G[x]

then forsome-integer y G[y]
and
forall-integer z if z < y
then not G[z]

ie

if a statement G[x] is true for some integer x, then there must be a least integer y for which it is true.

lessthan defined by not-exists z

forall-integer x,y x < y iff x ≤ y and not(x=y)

weak lessthan

forall-integer x x ≤ 0 iff x = 0

forall-integer x,
forall-positive y x ≤ y iff x=y or x ≤ y - 1

Prove:

forall-integer x,y x ≤ y iff forsome-integer z x + z = y

$x \le y$ iff $x|z = y$

$x=y$ or $x \le y|\Diamond$ iff $x|z = y$

$x=y$ or $x|\bullet \le y$ iff $x|z = y$

$x \neq y$ since $x|z$

$x|\bullet \le y$ iff $x|z = y$

let $z = \bullet$

===

===

Not Algebra

As stated, these rules do not incorporate the concepts of

1. arity -- addition (and multiplication) is defined as a collection of partitions of a number. The number of partitions is not constrained.

Example:

$\bullet\bullet\bullet\bullet\bullet\bullet = \bullet\bullet\bullet/\bullet\bullet/\bullet$ by partition removal

That is, partitions can be removed concurrently, any number at a time.

2. associativity -- there is no ordering of application of partition joining or removal

$\bullet\bullet\bullet \ \bullet\bullet \ \bullet = \bullet\bullet\bullet\bullet\bullet\bullet$ by concurrent joining

NOTE: partitioning removal and joining are the same thing.

3. commutativity -- removing a partition does not specify or require a privileged side.

4. zero -- there is no zero in this system. Zero, as absence, is absent.

NOTE: One is also the absence of change:

"subst ● for ● in A" = do not "subst ● for ● in A"

5. inverse:

a + ainv-a = <void>

● ◊ = <void>

ainv-a = "subst ◊ for * in A"

a * minv-a = ● "subst ● for A in A"

a * minv-a = "subst minv-a for ● in A"

"subst "subst ● for A in A" for ● in A" = ●

===

```
================================================================================
  Fig %%%: Modern Algebra of Rational Numbers

Additive Group
            x+e = x                         identity          e = 0
            x+x' = e                        inverse           x' = -x
            (x+y) + z = x + (y+z)           associativity
            x+y = y+x                       commutativity
            x = y  iff  x+z = y+z           decomposition theorem
Multiplicative Group
            x*e = x                         identity          e = 1
            x*x' = e                        inverse           x' = 1/x
            (x*y)*z = x*(y*z)               associativity
            x*y = y*x                       commutativity
            x = y  iff  x*z = y*z           decomposition theorem
Ring
            x(y+z) = x*y + x*z              distribute
================================================================================

================================================================================
Formal Description of Unit Ensemble Arithmetic

Fusion
            x|Δx = ><                       changeunit        Δx =
            x|y|z                           flat
            x = y  iff  x|z = y|z           decompose
Substitution
            [x e e] = x                     global            e = •
            [x x e] = e                     self
            [x • y • z]                     super-associativity
            [x • y] = [y • x]               commutativity
            x = y  iff  [x • z] = [y • z]   decompose
Distribution
            [x|y • w|z] = [x • w]|[y • w]|[x • z]|[y • z]
================================================================================

The primary differences between group theoretic arithmetic and unit ensemble
arithmetic are

      -- fusion does not have an identity
      -- flat fusion incorporates both associativity and commutativity
      -- substitution directly incorporates an inverse
      -- substitution is super-associative.

Each will be discussed in detail in the sequel.
```

UNIT AND BOUNDARY ARITHMETICS -- SECTION IVB October 1, 2008 10:55 PM
William Bricken
January 2007

UNIT ENSEMBLE ARITHMETIC
- Unit Arithmetic
  - Operations
  - Negative Numbers
  - Numerals as an Abbreviation
- Structure of Comparison of Wholes
  - Trichotomy
  - Comparison of Axioms and Theorems
    - No Zero
    - No Unit Successor
    - No Right/Left Theorems
  - §Conventional Axioms and Theorems of Comparison (Greater-Than)
  - Axioms and Theorems of Comparison for Solid Unit Ensembles
  - Total Ordering
    - Irreflexivity
    - Asymmetry
    - Transitivity

UNIT ENSEMBLE ARITHMETIC

We now map unit ensembles onto elementary arithmetic. Different wholes are named by different numerals; positive and negative integers are Solid and Unit units; addition and subtraction combine into fusion of different unit types; and both multiplication and division are substitution of ensembles for units. We will use the notation

<conventional-form> == <boundary-form>

to denote the transcription process in converting between systems. All such mappings are morphisms.

In Section %%%, unit ensembles were not associated with the concept of number, since the theory of unit ensembles can be interpreted in a wider scope. Here we map unit ensembles to integers, calling the theory unit ensemble arithmetic.

A number is denoted by a collection of identical marks, and computed by counting of marks. Counting is placing the units of a Whole into one-to-one correspondence with the sequence of integers. Conventional numerals are abbreviations for the "dot-pictures" that constitute the ensembles of unit arithmetic. For example, the numeral 5 is represented by the dot-picture ●●●●●.

-n	◊n	--typing or sign?--
...		
-3	◊◊◊	
-2	◊◊	
-1	◊	
0	no representation, void, ><	
1	●	
2	●●	
3	●●●	
4	●●●●	
...		
n	●n	

Figure : Representation of Integers as Ensembles

In Section %%%, we introduce various abbreviated notations for unit ensembles that are more tractable for manipulation. The structural and operational principles in a theory of unit ensembles depends upon the syntax; in fact, the interpretation as integer arithmetic is purely structural. Simplification is

lost when a notation incorporates unnecessary substructure. For example, unit ensembles apply to conventional integers only to the extent that the addition of two digits is taken as shorthand for fusion of ensembles. This shorthand can be extended to the addition and multiplication rules of digits less than the base of the numerical representation system. In our base-10 system, shifting from one order of magnitude to the next requires additional mechanism (carrying, borrowing, etc.) that further undermines the simplification provided by concept of fusion. The original form of addition, some 6,000 years ago was based on fusion-like principles that were abandoned in favor of base-10 positional notations. We keep the idea of a consistent base, but exchange identifying the power of a base by sequential position (place-value) for associating power with the idea of structural nesting (depth-value). We later provide succinct depth-value notations in base-2 and in base-10.

This section addresses the structural mechanisms of fusion and substitution based on pattern-matching of unit ensembles. By disassociating the particular representation of integers (but not of operations!) from the transformational theory, we can keep syntax much closed to the intended semantics of ensembles of identical objects in an identified space.

Operations

Addition is by direct composition (fusion), obeying the Additive Principle that the representation of a sum consists of the representations of its parts. Multiplication is by substitution of ensembles for units.

a+b	=def=	fuse(a|b)	a|b	place a and b in the same space, forming the fusion ab
e*a	=def=	sub(a,•,e)	[a • e]	substitute a for each • in e

Unit arithmetic has no zero; that is, the absence of an object "denotes" nothing. For fusion, there are no principles of ordering, of grouping, or of the arity of operations, since ensembles reside in a spatial representational space that allows all operations to occur in parallel. Substitution is a ternary operation, yet within this restriction, ordering and grouping of substitutions is also irrelevant. And there is no principle of induction. Finally, there are, as yet, no notational principles of grouping by magnitude or of place-value notation.

Examples

a = •••	called "3"
b = ••••	called "4"
a + b = •••|•••• = •••••••	called "7"

a * b = [●●●● ● ●●●] = ●●●●●●●●●●●● called "12"

Negative Numbers

Negative numbers provide an inverse for positive numbers, so that for every positive integer a,

$$a + -a = 0$$

The base case of the additive inverse is

$$1 + -1 = 0$$

We represent the positive unit as dot, ● , and the negative unit as diamond, ◊ . The unit ensemble analog of the base case of the Additive Inverse is then the Unit Annihilation Rule:

●|◊ = ● ◊ = void

The idea of using discrete unit types is incorporated in conventional representation for imaginary and real units, i and 1, but not for the two integer units 1 and -1. Instead, conventional systems have evolved using a unary operation, "negative", in conjunction with the positive integer 1 to generate the negative integer -1. This results in some clumsiness in the interaction of negative numbers with both subtraction and multiplication of negative units. In particular, changeunit (a substitution operation) operates on both types of ensemble units in the same manner.

Δ● = ◊
Δ◊ = ●

Unit Annihilation can be generated from positive unit identity and the definition of equality, and from negative unit identity and equality:

● = ●	◊ = ◊
●|Δ● = ><	◊|Δ◊ = ><
●|◊ = ><	◊|● = ><
●◊ = ><	◊● = ><

The unit ensemble definition of equality easily generalizes to any ensemble, regardless of type:

$$a|\Delta a = ><$$

In a void-based unit ensemble arithmetic, inverses annihilate into the absence of a representation, rather than generate an explicit representation (i.e. zero) that stands in place of the concept "nothing". Since all ensembles are ensembles of units, only the positive unit itself requires an inverse. In the

case of an ensemble, a, consisting of multiple units, the inverse is simply Δa. That is, instead of an inverse for every number, there is only one inverse, for the unit, while changeunit generalizes the idea of an unit inverse to any whole.

The mapping to unit integers is naturally

+1 == •
0 == ><
-1 == ◊

Sorts are mapped as follows:

P(a) == S(a) positive
N(a) == H(a) negative
I(a) == W(a) integer other than zero

0 == >< no zero

Solid ensembles are positive numbers; Hollow ensembles are negative numbers; and Whole ensembles are integers, with the exception that 0 does not have a unit ensemble representation. Zero is the absence of a mark.

The relationship between the Solid and Hollow units defines a Boolean-like domain, for which negation and changeunit have the same meaning:

not •a == Δ•a = ◊a
not ◊a == Δ◊a = •a

Note that (not •a) does not include the possibility that a=><. A typed ensemble is presumed to exist, so that its negation also exists. Void cannot be attributed properties, so that technically the expression (not void) is ill-formed. This position is somewhat philosophical, and various mathematical theories have encountered substantial difficulties distinguishing between existence and non-existent objects. For example, in order to avoid a non-existent intersection between disjoint sets, set theory includes an existent empty set. Unit ensembles, in contrast, are disjoint by definition, so that intersection is not defined. Fusion is similar to the union operation, but due to Unit Annilihation, fusion can effect operations similar to intersection.

There is no impact of two different unit types on the fusion operation, other than the necessity of the Unit Annihilation rule. Substitution is broadened to incorporate two regimes:

e*a == [a • e]

```
-e*a  ==  [a ◊ e]
```

The latter regime is value preserving, when seen as two applications of changeunit. The first application converts the Hollow unit for-form, which is as yet uninterpreted for multiplication, into a Solid unit which is characteristic of multiplication. The second global application restores the value of the result:

```
[a ◊ e] = Δ[a Δ◊ e] = Δ[a • e]
```

This then provides a syntactic definition for a Hollow for-form.

We now consider in detail the structure of integer operations (comparison, addition, subtraction, multiplication, division, exponents, fractions) in light of the theory of unit ensembles.

Numerals as an Abbreviation

Sometimes it is convenient to represent an ensemble in a short-hand notation. Rather than writing, or rather drawing, 8 units, •••••••• , we would prefer to simply write the numeral 8 to stand in place of the ensemble of 8 units. We will use this short-hand judiciously, since we do not want to include or to imply the conventional mechanisms of addition and multiplication. It is possible to express the rules of unit ensemble arithmetic, using numerals rather than unit dots, so long as we remember that such numerals must be composed as if they represented ensembles rather than convention numbers.

For example, the factors of 30 can be written concisely as

```
30 = 2*3*5
```

We can provide an explicit dot picture of this factorization using unit ensembles:

```
(•••••|•••••|•••••|•••••|•••••|•••••)           6 partitions of 5
(•••|•••|•••|•••|•••|•••|•••|•••|•••|•••)      10 partitions of 3
(••|••|••|••|••|••|••|••|••|••|••|••|••|••)  15 partitions of 2
```

However, this many units do not necessarily make the point of factorization clear. The short-hand notation using numerals is more helpful:

```
(5|5|5|5|5|5)                          6 partitions of 5
(3|3|3|3|3|3|3|3|3|3)                 10 partitions of 3
(2|2|2|2|2|2|2|2|2|2|2|2|2|2|2)       15 partitions of 2
```

```
===
OR
            30 = (15|15)
            15 = (5|5|5)

            30 = (10|10|10)
            10 = (5|5)

            30 = (6|6|6|6|6)
             6 = (3|3)
===

       (●●●|●●●|●●●|●●●|●●●|●●●|●●●|●●●|●●●|●●●)       10 partitions of 3

subst ● for ●●● in ^

        [● ●●● (●●●|●●●|●●●|●●●|●●●|●●●|●●●|●●●|●●●|●●●)]
      = (●|●|●|●|●|●|●|●|●|●)
      = (●●●●●●●●●●)

reverse is

      [●●● ● (●●●●●●●●●●)]    =30

      [●●● ● [● ●●● (●●●|●●●|●●●|●●●|●●●|●●●|●●●|●●●|●●●|●●●)]]

        [● ●● (●●●●●●●●●●)]
      = (●●●●●)

      [●● ● ●●●●●]            =10

      [●● ● [● ●● (●●●●●●●●●●)]]


      [●●● ● [●● ● ●●●●●]]    =30  (3*2*5)
```

Structure of Comparison of Wholes

==umbrella more up?==
Additionally, conventional relational structure is phrased with an implicative bias, constructing a distinction between the antecedent and the consequent. We prefer the biconditional form that asserts an equivalence between the components in a statement of symmetry or asymmetry.
===

The void-equivalent definition of equality that is insensitive to the unit type of a Whole:

$$a=b \text{ iff } a|\Delta b = ><$$

The definition uses inverse units to establish a one-to-one correspondence between units in a and those in b. Only under such a one-to-one correspondence of Solid and Hollow units will the resultant form be void-equivalent. In the simplest case:

$$a = b = \bullet$$
$$\bullet|\Delta\bullet = \bullet|\lozenge = \bullet\lozenge = ><$$

Absence of void-equivalence serves to define comparison operators, > and <, between two ensembles, regardless of unit type. Without loss of generality, we assume that S>H. The four possible cases are:

when ($\bullet$a and $\lozenge$b),	a>b,	$\bullet$.a|Δb
when ($\lozenge$a and $\bullet$b),	not a>b,	$\lozenge$.a|Δb = not $\bullet$.a|Δb
when ($\bullet$a and $\bullet$b),	a>b iff	$\bullet$.a|Δb
when ($\lozenge$a and $\lozenge$b),	a>b iff	$\bullet$.a|Δb

When two ensembles are of the same type, it is sufficient to apply changeunit to one and then fuse the results. The type of unit remaining after fusion defines the result of the comparison operation. When two ensembles are of a different type, if a is Solid, it is always larger, and if a is Hollow, it is always not larger. Combining these cases, we arrive at:

a>b iff $\bullet$.a|Δb greater-than

Trichotomy

The trichotomy principle for integers states that any given integer is either greater-than, equal-to, or less-than another.

if I(a) and I(b) then ((a > b) xor (a = b) xor (a < b))

The unit ensemble arithmetic analogue to the trichotomy principle is:

if W(a) and W(b) then $\bullet.a|\Delta b$
xor $a|\Delta b = ><$
xor $\diamond.a|\Delta b$

We elect to base the axioms of comparison on the conventional strict greater-than relation. As is well known, only one comparison operation need be axiomatized, the others are logical variations. In specific, given a definition of strict greater-than, >, and a definition of equivalence, =, then the other comparison relations are:

strict less-than	a < b	not a>b and not a=b
weak greater-than	a ≥ b	not a<b
weak less-than	a ≤ b	not a>b

Due to the non-existence of void,

not $\bullet a = \diamond a$

Therefore

not $\bullet.a|\Delta b = \diamond.a|\Delta b$

In the formulation of unit ensembles that follows, we use trichotomy to define three mutually exclusive comparison operations, listing them explicitly whenever a concept such as weak greater-than is needed.

The five comparison relations are thus defined as:

a > b	iff	$\bullet.a\|\Delta b$	greater-than
a ≥ b	iff	$\bullet.a\|\Delta b$ or a=b	weak greater-than
a = b	iff	$a\|\Delta b = ><$	equal
a ≤ b	iff	$\diamond.a\|\Delta b$ or a=b	weak less-than
a < b	iff	$\diamond.a\|\Delta b$	less-than

Comparison of Axioms and Theorems

Figure %%% presents the conventional axioms and theorems of a theory of comparison for positive integers. Fig %%% presents the axioms and theorems of

comparison for unit ensembles. Since Solid unit ensembles map directly onto positive integers, the two collections of rules are equivalent. Yet due to the difference in how each theory is formulated, the rules look significantly different. We address in particular the absence of zero, successor, and right/left structures in unit arithmetic.

==========
Figure %%%:
==========

No Zero

Unit ensembles have no zero, so that a zero rule does not occur. Should we transcribe the conventional right zero rule into the concepts of unit ensembles, we arrive at:

$$x > 0 \quad \text{iff} \quad x \neq ><$$

Structurally, the form (x > 0) is represented in unit arithmetic simply as x, while its type is •x. When addressing only positive ensembles (as the conventional axioms do here), a unit ensemble x is necessarily greater-than zero. Notationally, existence is confounded with not being void (zero in the interpretation). The ensemble x is trivially not equal to void as soon as we observe it. Rather than placing a definition of greater-than-zero into a token-string that must be read and understood, the spatial system calls upon visceral information, if there is an x, then it is greater-than there not being an x.

No Unit Successor

The right successor axiom defines greater-than in terms of a unit incremental increase in the number x. This comes from the conventional axiomatic perspective of defining numbers as unit increments, using a successor function. The disjunction

$$x > y \quad \text{iff} \quad x = y' \quad \text{or} \quad x > y'$$

is to prepare the definition for induction, x is either equal to one more than y (i.e. y'), or it is more than one greater. With this representation it is not possible to state that x is simply greater by some arbitrary amount, it is necessary instead to iterate through successors. Partially transcribing the right successor rule literally:

$$x > y \quad \text{iff} \quad x = y|\bullet \quad \text{or} \quad x > y|\bullet$$

Completing the transcription,

●.x|Δy iff x|Δ.y|● = >< or ●.x|Δ.y|●
●.x|Δy iff x|Δy|◊ = >< or ●.x|Δy|◊

The unit ensemble approach is not specifically limited to a unit increment. An ensemble a is greater-than an ensemble b whenever there are any units within a that are not replicated within b, regardless of how many. We would write

a > b iff a = b|n

Since void odes not participate in fusion, n must exist, N must be positive since

a = b|n iff ●.a|Δb|n

This changes the idea of successor from a sequential incremental approach, using the smallest unit increment, to a observational approach of looking for any additional units in one ensemble. Solid units in a and Hollow units in Δb are put in one-to-one correspondence, so that incremental counting is not necessary in order to determine whether or not there are remaining Solid or Hollow units after pairs of Solid and Hollow units annihilate. Of course, should a = b, then it is again obvious that nothing remains. So the concept of a successor is modified in two distinct ways: a difference between x and y can be of any magnitude, and that difference is determined by one-to-one correspondence rather than by incremental counting.

We can algebraically manipulate the unit ensemble definition to see the conventional definition of > in a different light:

a|Δ.b|● = >< or ●.a|Δ.b|●
a|Δb|Δ● = >< or ●.a|Δb|Δ●
a|Δb|◊ = >< or ●.a|Δb|◊
a|Δb|◊|● = ● or ●.a|Δb|◊
a|Δb = ● or ●.a|Δb|◊

The first form is trivially Solid, the second reverts back to the third line,

a|Δb|◊ = ><

Since we have shown that this equation is Solid, it follows that the second form on the last line is Solid. The disjunction is now no longer necessary, since we can generalize the specific statement that a is one larger than b to a statement that the type of unit left after fusion of a with Δb is Solid. That is, the disjunction is needed solely to provide a recursive definition so that

comparisons can be iterated over successors to arrive at a ground. However, the recursion is not needed when the definition of comparison can perform parallel one-to-one annihilations. All that is needed is

$$a > b \text{ iff } \bullet.a|\Delta b$$

or equivalently

$$a > b \text{ iff } a = b|c$$

The ensemble a is greater than the ensemble b whenever we can partition a into two existent parts, one of which is b. That is, we need only an existent c to demonstrate a>b. In the transcription of the conventional definition,

$$a|\Delta.b|\bullet = ><$$
$$a = b|\bullet$$
$$c = \bullet$$

The disjunction in the conventional definition extends c to be other than •, an extension that the unit ensemble definition incorporates directly.

The latter definition is in fact a direct transcription of the conventional axiom of right addition:

$$x > y \text{ iff } x = y + z$$

That is,

$$x > y \text{ iff } x = y|z$$

To summarize, the incremental right successor axiom can be converted into the right addition axiom when using unit ensembles because we do not need to distinguish between the successor function and the addition operation. The successor function is unnecessary because we do not need to invoke the Induction Principle to define addition as recursive incrementation.

No Right/Left Theorems

We do not need to distinguish between right zero (or right successor or right addition) and left zero/successor/addition because fusion does not support commutativity, or for that matter, any positional property of parts being fused together. Consider the literal transcriptions, with "not" interpreted as changeunit:

left zero: not 0>x

$\Delta_{\bullet}.><|\Delta x$

$= \Delta_{\bullet}\Delta x$ void is not fused

$= \Delta_{\Diamond} x$ x began Solid

$= \bullet x$ x ends Solid

Right zero asserts that x is Solid (Positive), and yes that is all it can be, since if it were void it would not exist.

left addition: x + y > x

$x|y > x$

$= \bullet.x|y|\Delta x$

$= \bullet y$

Left addition asserts that whatever we have added to x exists (again the conventional theorems are restricted to positive integers). We can see that both conventional theorems are built into the concepts of void and fusion.

===FIX

The right successor axiom and the left successor theorem,

right successor axiom: x > y iff x = y' or x > y'

left successor theorem: x' > y iff x = y or x > y

are structurally interchangable (not implicatively equivalent) in unit arithmetic:

==========

$x > y$ iff $x = y|\bullet$ or $x > y|\bullet$

$\bullet.x|\Delta y$ iff $x|\Delta.y|\bullet = ><$ or $\bullet.x|\Delta.y|\bullet$

$x|\Delta.y|\bullet = ><$

$x|\Delta y|\Delta\bullet = ><$

$x|\Delta y|\Diamond = ><$

$x|\Diamond = y$

$x|\bullet > y$ iff $x = y$ or $x > y$

$\bullet.x|\bullet|\Delta y$ iff $x|\Delta y = ><$ or $\bullet.x|\Delta y$

$x|\Delta y = ><$

$x|\Delta y|\bullet|\Diamond = ><$

$x|\bullet|\Delta.y|\bullet = ><$

$x|\bullet = y|\bullet$

Figure %%%:
==========

But more fundamentally, the incremental successor does not actually support a positional distinction. Transcribing

a|● > b iff a=b or a>b

●.a|●|Δb iff a|Δb = >< or ●.a|Δb

This is equivalent to saying

when a|Δb = ><, ●● or when ●.a|Δb, ●.a|●|Δb

The first case is the definition of the Unit Solid type, while the second case says that fusing a Solid Unit to a Solid ensemble does not change the Solid type, again just the definition of Solid.

Another approach is to use substitution of equals:

a|● > b iff a=b or a>b

when a>b, a|● > b

Both of these cases are trivially true. The first case above is simply the theorem,

when a=b, a|● > a

a|● > a
a|●|Δa > a|Δa
● > ><

===

Strict Greater-Than

Conventional Axioms and Theorems of Comparison for Positive Integers

x > 0 iff x ≠ 0	right zero
x > y iff x = y' or x > y'	right successor
x > y iff x = y + z	right addition
if x > y and y > z then x > z	transitivity
not x>x	irreflexivity
if x > y then not y>x	asymmetry
x' > x	adjacent
not 0>x	left zero
x + y > x	left addition
x' > y iff x = y or x > y	left successor

Axioms and Theorems of Comparison for Solid Unit Ensembles

a>b iff ●.a|Δb

a>b iff a = b|c

Total Ordering

What remains are the conventional relations that define a total ordering: irreflexivity, asymmetry and transitivity. Transcribing each:

not x>x	not ●.a|Δa
if x>y then not y>x	if ●.a|Δb then not ●.b|Δa
if x>y and y>z then x>z	if ●.a|Δb and ●.b|Δc then ●.a|Δc

Each of these, rather than being axiomatic to the concept of total ordering, is provable for unit ensembles. To do so, we need to interpret the logic connectives in terms of unit ensembles.

Irreflexivity

Irreflexivity for comparison of unit ensembles states that there will be no remaining units when an ensemble a is fused with another ensemble Δa composed of annihilating units. There is an obvious one-to-one correspondence, and under annihilation, the fused ensemble will vanish. This ==fix== is not a relational interpretation in which an object shares a particular relation (the comparison relation) with a replica of itself. Instead an ensemble is fused with a different ensemble that happens to be constituted of units that have been modified to be in one-to-one correspondence to their annihilators. This ==fix== is exactly the pigeon-hole principle, for which each pigeon (unit) occupies one cubby (annihilating unit). In comparing the two ensembles, the cardinality of each is never determined. As well, neither a nor Δa is iterated to compare units of each recursively, one at a time.

not $\bullet.a\|\Delta a$	
not $\bullet><$	$a\|\Delta a$ = void
TRUE	Void does not have a type

==clearer==
Relations abstractly map members of a set to those of the same or another set. Irreflexive relations have no mapping arrows connecting identical objects. In unit ensemble comparison, no ensembles are mapped onto themselves either. But there is no requirement that the set of possible ensembles be identified or mapped. Rather, the entire structure of the comparison relation is subsumed by the operation of fusion with annihilation.
===

Asymmetry

Orderings are not symmetric; the comparison relation forces directionality in the structure of the set of objects being compared. Thus,

$\bullet.a|\Delta b$ iff not $\bullet.b|\Delta a$

$\bullet.a|\Delta b$ iff $\lozenge.b|\Delta a$

The unit ensemble comparison operation is known not to be reflexive, which is to say that two ensembles being compared are not equal. Consider two cases. For the structure $a|\Delta b$ to be Solid, asymmetry asserts that the structure $b|\Delta a$ must be Hollow. Should we fuse a Solid ensemble with a Hollow ensemble, the one with the predominant unit types will prevail. When we fuse the two "asymmetric" structures,

$a|\Delta b|b|\Delta a = ><$

both are fully annihilated. Thus neither has a predominance of any type of unit. However, for either to have existed, at least one must have been Solid

and the other must have been Hollow. This is exactly the assertion of unit ensemble asymmetry, i.e. that one structure is Solid iff the other is Hollow. Again there is no consideration of cardinality and no iteration over set members.

===fix
Conventional asymmetry

a > b iff not b > a

is failure of the symmetry of unit annihilation. For the cases of •x or ◊x, a non-void form has somewhere been introduced. total ordering is accounting for non-void addition over every a>b. Non-void can be any c, not just •.
===

Transitivity

Again we convert the implicative statement of transitivity to one of biconditionality:

if •.a|Δb and •.b|Δc then •.a|Δc
•.a|Δb and •.b|Δc iff •.a|Δc

Here we have two structures that are known to be Solid. We also know that fusion of two Solid ensembles will also be Solid:

•.a|Δb|b|Δc = •.a|Δc

Thus we know that the result, a|Δc is necessarily Solid.

From the perspective of b, transitivity is saying

b|b = ><

which is true of all b, not just a specific b associated with a.

==========
Figure %%%: transitivity spatial form

a>b "&" b>c "iff" a>c

a|Δb "&" b|Δc "iff" a|Δc

where b|Δb = >< (use this appeal)

==========

===
Conventionally, transitivity applies across two instances of a relation when each share an argument. The common argument provides a "glue" that establishes the same relation between the two unmatched arguments. transitivity is of course central to the definition of a comparative relation, since the basis of the comparison is an ordering on the domain of arguments.

For unit ensembles, should we interpret the logical conjunction as another fusion...
===finish

William Bricken
January 2007

STRUCTURE OF UNIT ADDITION

We now compare the conventional axiomatic structure of addition to that of solid unit ensemble addition. Addition is transcribed as the fuse operation

$$x + y \;==\; a|b$$

with

$$I(x+y) \;==\; S(a|b)$$

Here the separator bar designates that we have begun by considering a and b to be in different spaces. Addition combines them into the same space via fusion. Thus, unit ensemble addition explicitly accounts for the act of designating and collecting ensembles that are to be added together. Once fused, units in a common space form a Whole.

A conventional interpretation iterates over units, for example

$$5 \;=\; 1+1+1+1+1$$

In unit ensemble arithmetic the decomposition of the Whole into unit parts is explicit rather than implicit as within a numeral such as 5:

$$5 \;=\; \bullet\bullet\bullet\bullet\bullet \;=\; \bullet|\bullet|\bullet|\bullet|\bullet$$

This permits the Whole to treated as an individual. All partitions into constituent wholes (i.e. parts) are on equal footing. The partition corresponding to iterated successors is the maximal partition of the Whole, but it is not privileged since any fused partition defines a whole. The set theoretic concept is that a Number is the class of sets with the same cardinality. Here, a Number is not only a cardinality, it is also the fusion of other Numbers that as parts sum to that cardinality. Associating a particular collection with a cardinal number, in both conventional and unit ensemble theories, is necessary for determining the unique value of the cardinality. However, in set theory, an infinity of classes of the same cardinality is presupposed, while in unit arithmetic Number is spatial structure, not numerical value. Cardinality is constructive, determined at the time of counting. Unit arithmetic is finite; not only does infinity not play a role, but an arbitrary large number, in order to be determined to exist, must be actively constructed via fusion from Numbers known to exist, or alternatively, must be assumed and then shown to partition into known Numbers. Unit ensemble arithmetic does not provide a ladder of inductions leading to infinity.

We adopt this perspective in order to study a theory of integers with concrete utility and intuitive structure, based in axioms that a young student would

understand. Here, a theory not designed to reach Cantor's paradise is an advantage rather than a weakness.

Unit Ensemble Axioms of Whole Number Arithmetic Addition		
Unary predicate	S(a)	positive integer sort
Variary function	fuse(a|...|z) = a...z	addition function
Generation	S(•) S(a|b)	
Uniqueness	a=b iff a|c = b|c	functional substitution
Decomposition theorem	a = • xor a = b|c	composition/decomposition

Conventional Axioms of Whole Number Arithmetic Addition

forall-positive-integer x, y:

Constant symbol (zero)			
	0		><
Unary function (successor)			
	x'		x|●
Unary predicate (positive integer)			
	I(x)		S(x)
Generation axioms			
	zero	I(0)	><
	successor	I(x')	S(x|●)
Uniqueness axioms			
	zero	x' ≠ 0	x|● ≠ ><
	successor	if x'=y' then x=y	x|● = y|● iff x=y
Addition function			
	right zero	x + 0 = x	x = x
	right successor	x + y' = (x+y)'	x|y|● = x|y|●
Theorems			
	right functional substitution	if x=y then z+x = z+y	x=y iff z|x = z|y
	sort	I(x+y)	S(x|y)
	right one	x+1 = x'	x|● = x|●
	commutativity	x+y = y+x	x|y = x|y
	associativity	(x+y)+z = x+(y+z)	x|y|z = x|y|z
	annihilation	if x+y=0 then x=y=0	x|y = >< iff x=y=><
	right cancellation	if x+z = y+z then x=y	x|z = y|z iff x=y
	decomposition	forsome-positive-integer z if x≠0 then x=z'	x ≠ ● iff x=z|●

Induction

forall-sentences F(x)
if F(0) and (if F(x) then F(x')) then F(x)

Comparison of Axioms and Theorems of Addition

The conventional axioms of addition are expressed here for positive whole numbers only. Although the tools of unit ensembles apply equally well to all integers, we address only positive integers, the Solid type of unit. Negative integers, corresponding to the Hollow unit type, are addressed in the next section on subtraction.

Following the observations from the axioms of comparison, we suppress analysis of right/left theorems and have included only one of the pair in the list of theorems for addition. We also do not transcribe the zero axioms, since they cannot be expressed in unit ensemble arithmetic. We abandon the concept of a successor function, as well as associativity and commutativity. And we move away from implicative definitions, in favor of algebraic definitions incorporating logical equivalence (iff).

The objects of the conventional theory of addition can be transcribed into unit ensembles:

0 == ><
x' == x|●
I(x) == S(x)

It is not possible to assert a zero generation axiom; fusion provides a more general unit ensemble generation axiom that does not require incrementation:

P(x') == ●.a|● = ●a|●●
P(x+y) == ●.a|b = ●a|●b

With no Hollow type, these read:

P(x') == a|● = a|●
P(x+y) == a|b = a|b

Similarly, there is not a zero uniqueness axiom. The axiom

x' ≠ 0

transcribes directly into

a|● ≠ ><

which is true by definition. Thus in unit ensemble arithmetic there is no additive identity. More accurately, the analogous unit ensemble axiom would not call upon void,

$a|\bullet \neq \bullet$

A direct transcription of the successor uniqueness axiom would be

if $x|\bullet = y|\bullet$ then $x=y$

The unit ensemble version is stronger in two respects. The fused constant can be of any magnitude, and the relationship is bidirectional.

$a=b$ iff $a|c = b|c$

In the conventional decomposition theorem, the existence of a z is asserted whenever x is not 0. This transcribes directly into

if $x \neq ><$ then $x = z|\bullet$

However, the decomposition theorem in unit ensemble arithmetic is the uniqueness axiom above. (This is not unexpected since decomposition is a theorem, not an axiom.) Decomposition is analogous to the exhaustive type theorem, which itself is a simple consequence of the sort definitions:

$a = \bullet$ xor $a = b|c$

The analogous statement of the conventional decomposition theorem is: This statement is true by virtue of the definition of fusion, i.e. the void cannot be fused.

$a \neq \bullet$ iff $a = b|c$

==========

show
$a=\bullet$ xor $a=/=b|c$ = $a=/=\bullet$ iff $a=b|c$

Figure %%%:
==========

Due to the absence of a successor function, the above axioms of unit ensemble addition already incorporate the addition function. That is,

$x+0 = x$	==	$a = a$	identity
$x+y' = (x+y)'$	==	$a\|b\|\bullet = a\|b\|\bullet$	identity

In summary, each of the conventional theorems for addition listed above is incorporated directly into the unit ensemble axioms and theorems.

Absence of Induction

Most surprisingly, we have not required an Induction Axiom as yet. For purposes of comparison, the unit ensemble induction axiom would read:

```
if F(0) and (if F(x) then F(x')) then F(x)
if F(●) and (if F(a) then F(a|b)) then F(a)
if F(●) and (not F(a) or F(a|b)) then F(a)
```

===

if Fa then F● & Fa->Fab

Fa iff F● & Fa iff Fab !?

===

STRUCTURE OF UNIT SUBTRACTION

We interpret the Hollow unit type, ◊, as negative one, -1.

-1 == ◊

With a negative unit, subtraction requires no additional mechanism, it is simply addition that is generalized over the two exclusive types of units.

We can also use changeunit to express a negative number:

-1 == Δ● = ◊

-a == Δa

In unit ensemble arithmetic, there is no distinction between the unary property of being negative, and the binary operator of subtraction:

a + b == a|b
a + -b == a|Δb
a - b == a|Δb

===
Changeunit is equivalent to a unary "negative", but subtraction is fusion. MORe
===

A negative number is added by fusing the changeunit of the positive number. Any number is subtracted also by fusing the changeunit of the number. Changeunit is the general operator that converts a given Whole into its additive inverse, its negative. These forms derive from the definition of equality:

a = b iff a - b = 0 == a = b iff a|Δb = ><

Note that these definitions are sufficient regardless of the unit types of both a and b:

(+a) - (+b) == ●a|Δ●b = ●a|◊b = a|Δb
(+a) - (-b) == ●a|Δ◊b = ●a|●b = ●.a|b
(-a) - (+b) == ◊a|Δ●b = ◊a|◊b = ◊.a|b
(-a) - (-b) == ◊a|Δ◊b = ◊a|●b = b|Δa

Subtraction Axioms

The conventional axioms of subtraction presented below are limited to positive numbers rather than to all integers. The accompanying transcriptions show that only annihilation is added to unit ensemble subtraction.

==========

Conventional Axioms of Positive Number Subtraction

for all positive-integer x,y,z:

predecessor	annihilation
(x+1)-1 = x	x|●|◊ = x
zero	identity
x-0 = x	x = x
successor	changeunit, annihilation
(x+1) - (y+1) = x-y	x|●|Δ.y|● = x|●|Δy|Δ● = x|●|Δy|◊ = x|Δy
decomposition	annihilation
x = (x-1) + 1	x = x|◊|●

==========

Unit ensembles include the features of subtraction directly, by establishing a notational identity between the operation of subtraction and the property of negativity of numbers. Conventionally these operator perspective and the property perspective are the same, however conventional concept and notation require that the two perspectives be treated differently, and then asserted to have the same value. In unit ensembles, there is no formal or conceptual difference in these perspectives, and this is supported by not being able to distinguish them notationally.

The idea of predecessor decomposition (N down to 0 in unit increments) is usually incorporated within a theory of subtraction, while successor composition (0 up to N in unit increments) falls under the theory of addition.
===say more===

UNIT AND BOUNDARY ARITHMETICS -- SECTION IVD October 1, 2008 10:55 PM
William Bricken
January 2007

STRUCTURE OF UNIT MULTIPLICATION

The interpretation of substitution-as-multiplication for unit arithmetic is:

```
  e*a   ==  [a • e]
-(e*a)  ==  [a ◊ e]
```

with

W([a • e]) and W([a ◊ e])

Put/In Symmetry

The interpretation for multiplication places only one additional structural constraint on the substitution operation, that of commutativity, or symmetry, between put-forms and in-forms:

[a • e] = [e • a] put/in symmetry

With commutativity of substitution, the semantics of substitution presented earlier provides all remaining conventional structural axioms of multiplication. The property that distinguishes the general concept of substitution from the concept of a multiplication operation is that in multiplication the put-form and the in-form commute without changing the value of the resultant Whole.

During substitution, we expect to find a replicate of the for-form within the in-form substitution environment. Put/in symmetry introduces a substantial modification to the idea of substitution: we expect also to find a replicate of the for-form within the put-form. Unit ensembles facilitate this symmetry since the for-form that defines multiplication is a single Solid unit, replicates of which constitute the parts of both the put-form and the in-form.
===say better ^===

Although the put-form and the in-form are symmetrical with regard to a result, the process of substitution does distinguish differential effort. Substitution-as-multiplication is symmetrical in space but not in the transformation steps required to implement the substitution. For example, consider the simple case of 2*3 vs 3*2:

a = ●●●
e = ●●

2*3 = [●●● • ●●] ==> ●●●|●●● ==> ●●●●●●

3*2 = [●● • ●●●] ==> ●●|●●|●● ==> ●●●●●●

2*3 requires two substitution actions, while 3*2 requires three substitutions. In both cases the multiple substitutions can be implemented in parallel. In effect, both cases require the same amount of time; however, 3*2 requires one more device (a "substitution-module") than does 2*3.

We can collapse this difference by making fusion blind to arity:

●●●|●●● = ●●|●●|●●

The additional effort associated with 3*2 comes from construction of three Wholes rather than two, as is the case of 2*3. Axioms of integer arithmetic do not usually address implementation costs, taking commutativity as an identity

3*2 = 2*3

Pragmatic differences occur only within implementation algorithms, a topic considered to be mathematically distinct from pure structure. With substitution-as-multiplication, these implementation issues cannot be avoided, since carrying out the substitution requires an implementation step.

==========

Conventional Axioms of Whole Number Arithmetic Multiplication

forall-positive-integer x, y, z:

right zero	in-void substitution
x*0 = 0	[>< • x] = ><
left fn substitution	in-equality
if x=y then x*z = y*z	[z • x] = [z • y]
right fn substitution	put-equality
if x=y then z*x = z*y	[x • z] = [y • z]
right successor	distribution, self substitution
x*(y+1) = x*y + x	[y|• • x] = [y • x]|[• • x] = [y • x]|x

Properties

Integer sort	Whole sort
I(x*y)	W([y • x])
left zero	in-void substitution
0*x = 0	[x • ><] = ><
left one	global substitution
1*x = x	[x • •] = x
right one	self substitution
x*1 = x	[• • x] = x
left successor	in-distribution, global substitution
(x+1)*y = x*y + y	[y • x|•] = [y • x]|[y • •] = [y • x]|y
commutativity	put/in symmetry
x*y = y*x	[y • x] = [x • y]
associativity	super-associativity
(x*y)*z = x*(y*z)	[z • [y • x]] = [[z • y] • x]
right distribute	put-distribution
x*(y+z) = x*y + x*z	[y|z • x] = [y • x]|[z • x]
left distribute	in-distribution
(x+y)*z = x*z + y*z	[z • x|y] = [z • x]|[z • y]

==========

==========

Unit Ensemble Axioms of Whole Number Multiplication

Whole sort

W([a • e])

put/in-symmetry

[a • e] = [e • a]

put/in-void substitution

[>< • e] = [a • ><] = ><

self/global substitution

[x • •] = [• • x] = x

put/in-equality [a • e] = [b • e] iff a=b

super-associativity

[a • [c • e]] = [a [• c •] e] = [[a • c] • e]

put/in-distribution

[a|b • e|f] = [a • e]|[b • e]|[a • f]|[b • f]

Hollow Units

Solid/Hollow sorts

S(•)
H(◊)
W([a ◊ e])

changeunit

[a • e] = Δ[a ◊ e]

==========

Comparison of Axioms and Theorems of Multiplication

The axiom comparison table for multiplication includes void and right/left theorems since they have corresponding unit ensemble substitution forms. This table does not include conventional axiomatization of multiplication by negative numbers, however the axioms of multiplication for unit ensembles cover integers in general.

Each of the rules of substitution for put/in-forms, developed earlier independently of the interpretation for integers, occurs in the table of mappings. Those rules addressing the for-form do not; for-equality and for-void substitution are addressed under the interpretation for division.

Value maintenance and distribution of equality also do not occur within the axioms of multiplication. Instead, these rules are part of the deductive sub-structure of all axioms. It is for this reason that substitution-as-multiplication can be deeply integrated into the algebraic structure of arithmetic.

The substitution rules found in predicate calculus are shown below in a more limited form for which the generalized for-form is replaced by a unit, for interpretation as multiplication:

$[a \bullet e] = e \text{ iff } a = \bullet$	value maintenance
$[a \bullet e=f] = [a \bullet e]=[a \bullet f]$	distribution over equality

The general idea of value maintenance (substitution of equals) degrades into an application of self substitution. The general idea of distribution over equality remains an application of functional substitution.

Commutativity as the Pivot

The axiomatic structure of substitution-as-multiplication differs from that developed without an interpretation for integers by only one additional rule, that of put/in symmetry, which is interpreted as the commutativity of multiplication. This rule is pivotal since it alone converts generic substitution into a model for multiplication. Thus, contrary to conventional axiomatizations which prove commutativity as a theorem, we take it as fundamental. All varieties of left/right rules can be seen as application of put/in symmetry, as can the special rules of void, self, and global substitution. Further, put/in symmetry provides the necessary basis for super-associativity, which permits virtually free exchange of put-forms and in-forms over any composition of substitutions.

from the point-of-view of implementation, symmetry transformations are directional steps in a temporal computational process that can be precompiled. That is, the put-form and the in-form can be pre-ordered for preferred computational sequences. Thus, right and left substitution functions are necessary for pragmatics but not for axiomatization.

For example, the "left one" rule (i.e. the multiplicative identity) corresponds to global substitution,

$$1*x = x \qquad [x \bullet \bullet] = x$$

while the "right one" rule corresponds to self substitution,

$$x*1 = x \qquad [\bullet \bullet x] = x$$

Both are equivalent under the commutativity of substitution:

$$[x \bullet \bullet] = [\bullet \bullet x] = x$$

Three other pairs of substitution rules combine under commutativity of substitution. Put-void substitution and in-void substitution correspond respectively to right and left zero rules:

$$[>< \bullet x] = [x \bullet ><] = ><$$

Put-equality and in-equality correspond to right and left function substitution:

$$[x \bullet z] = [y \bullet z]$$
$$[z \bullet x] = [z \bullet y]$$

Put-distribution and in-distribution correspond to the right and left successor rules:

$$[y|\bullet \bullet x] = [y \bullet x]|[\bullet \bullet x]$$
$$[y \bullet x|\bullet] = [y \bullet x]|[y \bullet \bullet]$$

The conventional successor model is more restricted than the unit ensemble model since addition and multiplication rules are defined in terms of unit increments. In the restricted case, put- and in-distribution collapse by self/global substitution:

$$[y|\bullet \bullet x] = [y \bullet x]|x$$
$$[y \bullet x|\bullet] = [y \bullet x]|y$$

In unit ensemble arithmetic, successors are not used, they are generalized to distribution of fusion over substitution. The more general form of put/in fusion distribution over substitution:

$$[a|b \bullet e|f] = [a \bullet e]|[b \bullet e]|[a \bullet f]|[b \bullet f]$$

Associativity

We have seen that associativity of substitution can take many forms. For the particular interpretation of substitution-as-multiplication, together with put/in commutativity, associativity takes on a limited form.

$$[a \bullet [b \bullet e]] = [[a \bullet b] \bullet e]$$

All variables are in an odd positional index; they stand in place of arbitrary forms. Unit for-forms are in all even indexed positions.

With regard to the interpretation, substitution can be in any order. That is, we can substitute b for units in e, and then substitute a for units in the result. These units can be provided only by b, since the first substitution replaced all units in e by b. Alternatively, we can substitute a for units in b, and then substitute that result for units in e.

Here is an example:

a = ●● b = ●●● e = ●●●●

[●● • [●●● • ●●●●]] = [●● • ●●●●●●●●●●●●] = ●●●●●●●●●●●●●●●●●●●●●●●●
[[●● • ●●●] • ●●●●] = [●●●●●● • ●●●●] = ●●●●●●●●●●●●●●●●●●●●●●●●

This transcribes as

2*(3*4) = (2*3)*4 = 24

Note again the pragmatic difference in implementation. 2*12 is well-suited for a bitwise binary implementation, whereas 6*4 is better wuited for a half-byte implementation.

Multiplication of Unit Types

We now consider in detail the multiplication of ensembles of differing types. The goal is to understand the difference of interpretation between

$$e*a \;==\; [a \bullet e] \quad \text{and} \quad -e*a \;==\; [a \lozenge e]$$

and to understand the special interpretation of failure-to-match-units, which is to force a match via changeunit and then adjust the change in value by another application of changeunit to the result. This special interpretation assumes that both the for-form and the in-form are pure, consisting of one type of unit only. We have assured this because fusion incorporates Unit Annihilation. We do not address here operations on partitions, prior to fusion, which may contain both unit types. However, changeunit is defined to address both unit types and thus applies to any partitioned or whole form.

We begin with some examples:

```
sa = ●●●          ha = ◊◊◊
se = ●●           he = ◊◊

 se*sa =  3*2      ==   [●●● • ●●] = ●●●●●●
 se*ha = -3*2      ==   [◊◊◊ • ●●] = ◊◊◊◊◊◊
-he*sa = -(3*-2)   ==   [●●● ◊ ◊◊] = ●●●●●●
-he*ha = -(-3*-2)  ==   [◊◊◊ ◊ ◊◊] = ◊◊◊◊◊◊
```

Hollow Unit Substitution

The interpretation of [a ◊ e] for multiplication, i.e. "substitute a for ◊ in e", is directly derivable from the definition of substitution-as-multiplication, [a • e], using two applications of changeunit, which preserves the value of the substitution:

```
[a ◊ e] = Δ[a Δ◊ e] = Δ[a • e]
```

Consider two cases:

```
when ◊[a ◊ e],  •[a • e]  ==    e*a
when •[a ◊ e],  ◊[a • e]  ==  -(e*a)
```

In the first case, when the substitution [a ◊ e] results in a Hollow ensemble, changeunit converts it into a Solid ensemble, which is interpreted as e*a. When the substitution results in a Solid ensemble, changeunit converts it into a Hollow ensemble, which is then interpreted as -(e*a). These interpretations are thus independent of the unit type of both a and e.

Failure to Match Units

Failure to identify a unit match for substitution, such as in [a • ◊] is handled in the same manner as is [a ◊ e], that is by two applications of changeunit:

[a • ◊] = Δ[a Δ• ◊]] = Δ[a ◊ ◊] = Δa
[a ◊ •] = Δ[a Δ◊ •]] = Δ[a • •] = Δa

The use of changeunit assures that a substitution never fails to match.

In each of the four interpreted examples in the prior section, the types of the for-form and the in-form match. There are four other possibilities for which the for-form does not occur within the in-form:

-se*sa = -(3*2) == [••• ◊ ••] = Δ[••• • ••] = Δ•••••• = ◊◊◊◊◊◊
-se*ha = -(-3*2) == [◊◊◊ ◊ ••] = Δ[◊◊◊ • ••] = Δ◊◊◊◊◊◊ = ••••••
he*sa = 3*-2 == [••• • ◊◊] = Δ[••• ◊ ◊◊] = Δ•••••• = ◊◊◊◊◊◊
he*ha = -3*-2 == [◊◊◊ • ◊◊] = Δ[◊◊◊ ◊ ◊◊] = Δ◊◊◊◊◊◊ = ••••••

These failure-to-match-units cases invoke forced-matching via changeunit, and thus permit the substitution to proceed. These examples illustrate how multiplication of signed integers is incorporated into substitution. We now examine substitution over different unit types in depth.

Substitution over Different Unit Types

Consider all eight possible cases of unit substitution:

[• • •]
[• • ◊]
[◊ • •]
[◊ • ◊]
[• ◊ •]
[• ◊ ◊]
[◊ ◊ •]
[◊ ◊ ◊]

There are three types of structure within these cases, natural substitution, put/in symmetry, and failure-to-match-units:

Natural Substitutions

Four substitutions have a natural interpretation of replacing occurrences of the for-form within the in-form. In these, the type of the for-form matches the type of the in-form. For example, the first substitution below calls for the

(trivial) operation of replacing all occurrences of the for-form ● within the in-form ● by the put-form ●.

[● ● ●] = ●
[● ◊ ◊] = ●
[◊ ● ●] = ◊
[◊ ◊ ◊] = ◊

Each of these cases follows the pattern of global substitution,

[A E E] = A

while two also follow the pattern of self substitution,

[A A E] = E

Put/in Symmetry Identities

[● ● ◊] = ◊
[◊ ◊ ●] = ●

These two cases have the structure of failure-to-match-units, as well as the structure of self substitution. We demonstrate that the forced-matching procedure is consistent with values expected from put/in symmetry.

[● ● ◊] = Δ[● Δ● ◊]
= Δ[● ◊ ◊]
= Δ●
= ◊

Interpreting put/in symmetry as commutativity of multiplication, we would expect

[● ● ◊] = [◊ ● ●] = ◊
[◊ ◊ ●] = [● ◊ ◊] = ●

The result of this substitution is consistent with that derived using changeunit. For these two cases, forced-matching, self substitution, and put/in symmetry all behave in the same manner.

Unavoidable Failure-to-Match-Units

The remaining two cases,

[● ◊ ●]

[◊ ● ◊]

obey commutativity by default, so that in all cases, put/in symmetry holds. These two cases, though, could generate a potential model contradiction. Due to failure-to-match-units, we would intuitively expect the in-form to remain the same:

[● ◊ ●] = ●
[◊ ● ◊] = ◊

For example, substituting the put-form ● for the for-form ◊ fails since there are no instances of ◊ within the in-form ●. A failed substitution would leave the in-from unchanged. However, under the interpretation for multiplication, we have

(-1)(-1) == [◊ ● ◊] = ●
-(1)(1) == [● ◊ ●] = ◊

These two cases are brought into consistency by removing the possibility of failure-to-match-units using forced-matching via changeunit. Two applications of changeunit maintains value:

[● ◊ ●] = Δ[● Δ◊ ●]
= Δ[● ● ●]
= Δ●
= ◊

Similarly,

[◊ ● ◊] = Δ[◊ Δ● ◊]
= Δ[◊ ◊ ◊]
= Δ◊
= ●

Finally we could examine the value of [● ◊ ●] algebraically. In general,

[a [a c e] e] = c

[a [a c e] e]
= [[a a c] e e]
= [c e e]
= c

Therefore,

[● [● ◊ ●] ●] = ◊

Consider the middle substitution structure,

either [• ◊ •] = • or [• ◊ •] = ◊

Case 1: [• ◊ •] = •

By substitution, we find that case 1 generates a contradiction.

[• [• ◊ •] •] = [• • •] ≠ ◊

Case 2: [• ◊ •] = ◊

By substitution, case 2 generates a consistent solution

[• [• ◊ •] •] = [• ◊ •]
= ◊

The net result of this analysis is that the single changeunit rule for Hollow substitution is sufficient to incorporate the multiplication rules for positive and negative numbers.

Factoring Whole Numbers

Whole numbers are either prime or composite. We will not address prime numbers directly; indirectly a number is prime if it is not composite. Factors of whole numbers identify the ways numbers are composed by multiplication. In contrast to fused partitions (which identify ways in which whole numbers are composed by addition), the factors of a number are unique. Partitions are decompositions of a fusion, we now explore decompositions of a substitution.

For example, the number 6 has the same value as the product 2*3. In ensemble notation,

6 == •••••• = [• • ••••••] = [•••••• • •]

2*3 == [••• • ••]

An ensemble can always be written as self substitution of units, and as a global substitution of a unit, equivalent to 1*6 and 6*1 respectively.

Factoring is then an inverse of self/global substitution:

2*3 = 1*6 == [••• • ••] = [•••••• • •]

Diagrammatically, the forms in the put-form position and the in-form position are factors for a substitution having a unit in-form (alternatively, put-form).

Factors identify equal partitions,

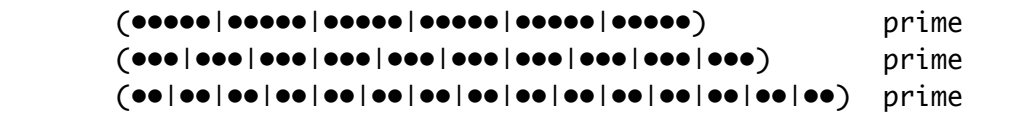
●●●●●● = (●●●|●●●) = (●●|●●|●●)

It is not necessary to "count" the number of times that a part "divides-into" a whole, rather it is sufficient to identify the prime factor equal-part partitions, those that cannot be partitioned further (excluding maximal unit partitioning).

Factor Trees

Decomposition of integers into prime factors requires successive factoring of component factors until all are prime. Fig %%% shows a factor tree for the integer 60.

Figure %%%: factor tree for 60 --also on page 11--



Two different partial factorings can be shown to be equal to the same integer simply by reducing both to prime factors. For example,

4*15 = 10*6 = 60

We will show how successive factoring can be achieved within the substitution-as-multiplication structure. When

[a • e] = [b • f]

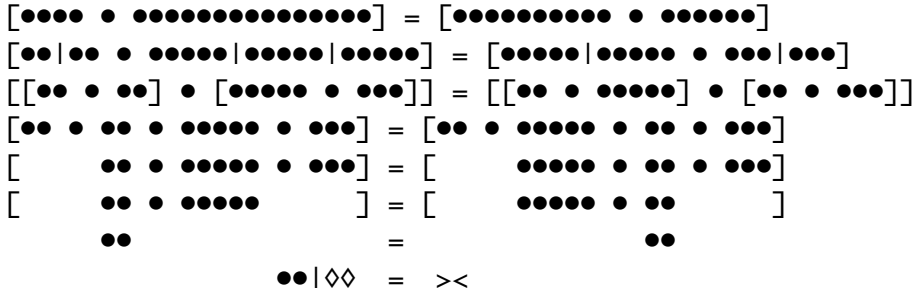
we can apply the fusion definition of equality,

[a • e]|Δ[b • f] = ><
[a • e]|[b ◊ f] = ><

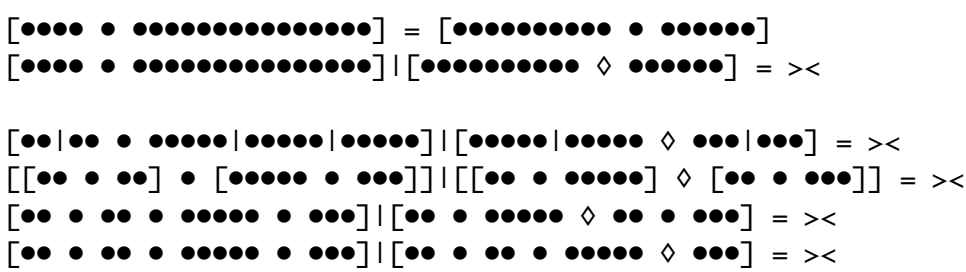
To solve the void-equivalence, we need each substitution to annihilate. This requires the substitutions to be fused at the put-form or at the in-form. To

fuse the substitutions, we need two of the three substitution forms to be identical. We illustrate this process with an example, 4*15 = 10*6.

The first method uses put-equality:



The second method uses direct fusion:



===FINISH?===

Generally

if a*e = b*1 then a*1 = b/e and b*1 = a/e

if [a • e]=[b • •] then [a • •]=[b e •] and [b • •]=[a e •]

The substitution structure of division is presented in the next section.

STRUCTURE OF UNIT DIVISION

Multiplication calls for substitution of an ensemble *a* for each unit in a second ensemble e. Division is the inverse of multiplication, calling for substitution of *a* single unit for each entire put-form that can be partitioned out of the in-form. Replacing units by an ensemble defines multiplication; replacing ensembles by units defines division. The asymmetry of divide is incorporated in the relationship between the put/in-forms and the for-form within substitution.

```
e*a  ==  sub(a,•,e)          -(e*a)  ==  sub(a,◊,e)

e/a  ==  sub(•,a,e)          -(e/a)  ==  sub(◊,a,e)
```

so that

```
[a • [• a e]] = e            [a ◊ [◊ a e]] = e
[• a [a • e]] = e            [◊ a [a ◊ e]] = e
```

An example:

```
a = ••
e = ••••••
e/a  ==  [• •• ••••••] = •••
```

The unit ensemble model implements division via partitioning. In the example, all •• forms within •••••• are identified, and each replaced by a unit.

```
e = •••••• ==> (••|••|••)

  [• •• ••••••]
= [• •• ••|••|••]
= [• •• ••]|[• •• ••]|[• •• ••]
= •|•|•
= •••
```

We first examine the conventional structures of exact divide, quotient, and remainder, as they relate to the substitution operation. We then discuss the impact of the interpretation as division on the concept of substitution, and finally show that quotient and remainder functions are convenient names for aspect of the division process, but are not necessary for a definition of unit ensemble division.

The Divide Function

The conventional definition of the divide function incorporates two component definitions, quotient and remainder:

quotient(y,x) = y//x == q[• x y]

remainder(y,x) = rem(y/x) == r[• x y]

We provide shorthand notation for the conventional functions, and specialized substitution forms for the unit arithmetic functions. The substitution forms are minor modifications of general substitution-as-multiplication for which some special cases have special reductions. These special substitution operations are represented by square brackets with a prefix label.

The definition of the divide function, incorporating quotient and remainder, is:

y/x = q + r/x iff y = x*q + r and q = y//x and r = rem(y/x)

Transcribing this into substitution notation:

[• x y] = q|[• x r] iff y=[x • q]|r and q=q[• x y] and r=r[• x y]

The unit arithmetic representation can be condensed by expanding the specialized q and r substitution operations:

[• x y] = [• x q[• x y]|r[• x y]] iff y = [x • q[• x y]]|r[• x y]

The quotient function defines exact division whenever the remainder is zero. The remainder function contributes to the definition of a proper fraction. We will later show that the unit arithmetic definition of divide does not require fragmentation into an exact division with a possible remainder.

Quotient and Remainder

The quotient and remainder functions subdivide the divide function into an exact division and a remainder. In the case that an ensemble cannot be partitioned into parts all equal to a given whole (the divisor), a remainder will be present. Consider for example the unit ensemble substitution structure corresponding to 7/2:

7 == ●●●●●●●

7/2 == (●●|●●|●●|●)

We first partition the "numerator" ensemble into wholes that are equivalent to the "denominator" ensemble. This can be a parallel process. The algebraic computation of the division proceeds as follows:

```
  [• •• •••••••]                                  division-as-substitution
= [• •• ••|••|••|•]                               partition the in-form
= [• •• ••]|[• •• ••]|[• •• ••]|[• •• •]          distribute the fusion
= •|•|•|[• •• •]                                  apply global substitution
= •••|[• •• •]                                    fuse the result
```

Division is a substitution process within which an in-form (or a put-form by put/in symmetry) is partitioned, distributed, and reduced. The fused result will consist of a whole (called the quotient) and possibly an incipient substitution (called the remainder) that represents a proper fraction.

The above example simplifies to a partition consisting of a whole and a substitution that cannot be carried out. In general, division-as-substitution partitions an ensemble into a whole which counts the partitions of an in-form into a for-form, and a remaining substitution for which the for-form is larger than the in-form. Algebraically,

```
y = x*q + r
```

becomes

```
y = [x • q[• x y]]|r[• x y]
```

which can be read as

```
<in-form> = [<for-form> • <quotient-whole>]|<incipient-substitution>
```

The conventional axioms of the quotient function include by a base case and a recursive model of successive addition (or subtraction) of the divisor that counts the addition steps. We include the literal transcription to unit arithmetic for comparison.

```
if x>y then y//x = 0          •.x|Δy  iff  q[• x y] = ><
(x+y)//x = y//x + 1           q[• x y|x] = q[• x y]|•
```

The conventional axioms of the remainder function include by a base case and a recursive model of successive addition (or subtraction) of the divisor that ignores the addition steps.

```
if x>y then rem(y/x) = y      •.x|Δy  iff  r[• x y] = y
rem((x+y)/x) = rem(y/x)       r[• x y|x] = r[• x y]
```

Whenever the number being divided is less than the divisor, we have a proper fraction. In this case the quotient function is zero, while the remainder function returns the numerator of the proper fraction.

Should the number being divided be larger than than the divisor, then the divisor is subtracted once, the division count is incremented by one, and the division process recurs. The remainder in this case is adjusted to reflect the new number being divided.

The unit arithmetic transcription of the quotient function shows that division is an application of fusion distribution followed by global substitution.

$$q[\bullet\ x\ y|x] = q[\bullet\ x\ y]|q[\bullet\ x\ x] = q[\bullet\ x\ y]|\bullet$$

Thus far quotient substitution and unrestricted substitution are the same. The base case modifies an incipient substitution to be void

$$\bullet.x|\Delta y \quad \text{iff} \quad q[\bullet\ x\ y] = ><$$

We can embed the less-than condition for the quotient base case directly into the substitution by comparing the substitution to unity:

$$x>y \quad \text{iff} \quad y/x < 1$$

This permits a type check to determine the base case for division-as-substitution:

$$y/x < 1 \quad == \quad \Diamond.[\bullet\ x\ y]|\Delta\bullet = \Diamond.[\bullet\ x\ y]|\Diamond$$

The comparison x>y is true whenever y/x is less than 1.

$$\begin{aligned}
&[\bullet\ x\ y]|\Diamond \\
=\ &[\bullet\ x\ y]|[\bullet\ \bullet\ \Diamond] \\
=\ &[\bullet\ x\ y]|[\bullet\ [\bullet\ x\ x]\ \Diamond] \\
=\ &[\bullet\ x\ y]|[\bullet\ [x\ x\ \bullet]\ \Diamond] \\
=\ &[\bullet\ x\ y]|[\bullet\ x\ [x\ \bullet\ \Diamond]] \\
=\ &[\bullet\ x\ y|[x\ \bullet\ \Diamond]] \\
=\ &[\bullet\ x\ y|\Delta x]
\end{aligned}$$

Thus, using the decomposition version of division iteration,

$$\begin{aligned}
&\text{if } \Diamond.[\bullet\ x\ y|\Delta x] \\
&\qquad q[\bullet\ x\ y] = >< \\
&\qquad r[\bullet\ x\ y] = y
\end{aligned}$$

```
            else        q[• x y] = q[• x y|Δx]|•
                        r[• x y] = r[• x y|Δx]
```

Conventional Axioms for Divide, Quotient and Remainder Functions

divide

```
      y/x = z + r/x  iff  y = x*z + r  and z = y//x  and r = rem(y/x)

      [• x y] = z|[• x r]  iff  y=[x • z]|r  and  z=q[• x y]  and r=r[• x y]

      [• x y] = [• x q[• x y]|r[• x y]]  iff  y = [x • q[• x y]]|r[• x y]
```

less-than quotient

```
      if x>y then y//x = 0          •.x|Δy  iff  q[• x y] = ><
```

one quotient*

```
      if x=y then y//x = 1          x|Δy = ><  iff  q[• x y] = •
```

addition quotient

```
      (x+y)//x = y//x + 1           q[• x y|x] = q[• x y]|•
```

less than remainder

```
      if x>y then rem(y/x) = y      •.x|Δy  iff  r[• x y] = y
```

zero remainder*

```
      if x=y then rem(y/x) = 0      x|Δy = ><  iff  r[• x y] = ><
```

addition remainder

```
      rem((x+y)/x) = rem(y/x)       r[• x y|x] = r[• x y]
```

Exactly Once

In the base case where the number being divided is equal to the divisor, the quotient function returns one while the remainder function returns zero.

one quotient*
if x=y then y//x = 1 x|Δy = >< iff q[• x y] = •

zero remainder*
if x=y then rem(y/x) = 0 x|Δy = >< iff r[• x y] = ><

Iterative Decomposition

Here we have expressed the division of y by x incrementally. The quotient of the sum x+y when divided by x is one more than the quotient of y divided by x. The number x+y is consequently reduced to y,

(x+y) - x = y

The remainder function for x+y is the same as the remainder function for y.

In substitution semantics, the quotient is simply a distribution of fusion of the substitution. Subtraction does not enter into the definition, it is converted into a global substitution of a unit for x:

q[• x y|x] = q[• x y]|q[• x x] = q[• x y]|•

Thus, with respect to distribution over fusion, substitution and the specialized quotient function are identical. The difference is in the proper fraction base case, for which the quotient function terminates with void rather than returning the proper fraction. This component is instead returned by the remainder function. Thus, symmetrically, the remainder function returns void rather than the unit under global substitution:

r[• x x] = ><

r[• x y|x] = r[• x y]|r[• x x] = r[• x y]

From this we can see that the one-quotient and zero-remainder rules are theorems derivable from a variant definition of global substitution:

q[===finish

We discuss the axioms of division after making some observations about the substitution structures of division.

Exact Division

We discuss the interpretation of unit ensembles for divides, quotient, and remainder functions, concentrating first on the simplest case when the remainder of a division is zero. Proper fractions, for which the in-form is smaller than the for-form, are considered in a following section.

When the in-form is larger than the for-form, division can be interpreted to take place by the conventional technique of successive subtraction. An ensemble, e, is exactly divisible by another ensemble, c, when it is possible to partition e into parts that all match the for-form. That is, exact division is defined by partitioning a whole into equal parts. Unit partitioning, for which each partition consists of one unit, is the maximal partition characteristic of all unit ensembles that defines the number of the whole. Maximal partitioning corresponds to division by 1, while no partitioning corresponds to division by the number itself. A prime number, of course, is one for which the divides function always includes a remainder, excluding maximal and no partitioning.

Substitution-as-Division

The interpretation of substitution-as-division casts new light on the substitution-as-multiplication process. The substitution rules interpreted for multiplication do not change, however put-void substitution must be undefined under the interpretation for division:

undefined == [a >< e]

The reason this substitution is undefined, from the perspective of unit ensembles, is that the for-form must exist in order to identify a match within the in-form. Although void-equivalent forms are permitted within substitutions, no void-equivalent forms are permitted under the interpretation for integers.

The axiomatic structures of global and self substitution find explanation when interpreted as division:

e*a/e = a == [a e e] = [e e a] = a
e*a/a = e == [e a a] = [a a e] = e

Self substitution is simply multiplying the in-form by 1 (i.e. a/a). Global substitution is multiplying the put-form by 1. Put/in symmetry unifies these two apparently different forms of substitution.

Reciprocal Symmetry

The substitution operation incorporates a model of division as well as multiplication. There is reciprocal symmetry between the put-for and the for-form. Substituting a for c in e combines multiplication and division into one function:

```
e*a/c  ==  [a c e]
```

Put/in symmetry is commutativity of multiplication, while put/for symmetry is reciprocality of division.

```
a/c  ==  [a c •] = [• c a]
```

A unit reciprocal is constructed by changing the place of the put-form with the for-form.

```
a/1  ==  [a • •] = [• • a] = [a ◊ ◊] = [◊ ◊ a]
1/a  ==  [• a •] = [◊ a ◊]
```

These definitions are positive when the two unit forms participating in the substitution are of the same type, and negative when they differ.

```
-a/1  ==  [a • ◊] = [◊ • a] = [a ◊ •] = [• ◊ a]
-1/a  ==  [• a ◊] = [◊ a •]
```

Reciprocals

The substitution structure for a reciprocal is

```
1/c  ==  [• c •]
```

Replacing an even indexed form directly by its reciprocal is improper, since this substitution shifts the even indexed form into an odd index position. The proper form is to construct the multiplication of odd index forms by the reciprocal, that is:

```
[a c e] = [a • [[• c •] • e]]
        = [a • [• [c • •] e]]
        = [a • [•  c      e]]
        = [a [• •  c]     e ]
        = [a       c      e ]
```

In order to construct a proper substitution form, the reciprocal for-form must be bracketed with one or the other of the put/in-forms, thus the reason for the

second set of inner brackets. However, within the flat substitution notation this is a moot point, since

```
[a • [[• c •] • e]] = [a  • •  c • •  e]
                    = [a  • • [c • •] e]
                    = [a  • •  c      e]
                    = [a [• •  c]     e]
                    = [a       c      e]
```

A Tighter Interpretation

Substitution for a Hollow for-form was previously interpreted as converting a multiplication result to its negative:

```
-(e*a)  ==  [a ◊ e]
```

This interpretation is now no longer needed; instead it is a consequence of the interpretation for division, specifically division by -1:

```
(e*a)/(-1)  ==  [a ◊ e]
```

We then have a quite powerful mechanism for proving substitution equivalence. The interpretation of unit substitutions becomes:

```
(1*1)/1     ==  [• • •] = •
(1*-1)/1    ==  [◊ • •] = ◊
(-1*1)/1    ==  [• • ◊] = ◊
(-1*-1)/1   ==  [◊ • ◊] = •

(1*1)/-1    ==  [• ◊ •] = ◊
(1*-1)/-1   ==  [• ◊ ◊] = •
(-1*1)/-1   ==  [◊ ◊ •] = •
(-1*-1)/-1  ==  [◊ ◊ ◊] = ◊
```

That is, an odd number of Solid forms within a substitution structure gives a Positive result, while an even number gives a Negative result.

Super-associativity of Substitution

The associative rule for substitution as multiplication,

```
(e*b)*a = e*(b*a)

[a • [b • e]] = [[a • b] • e]
```

can be generalized to include substitution as division:

[a c [b d e]] = [[a c b] d e]

which transcribes as

(a/c)*(b*e/d) = (a*b/c)*(e/d)

We might also note that the equality can be rewritten as

[a [c b d] e]

which is

(a*e)/(c*d/b)

That is, given a nested composition of substitutions, the interpretation does not change when the substitution is shifted over the arguments to the right or to the left. When this is combined with put/in symmetry, and with variable reordering derived from associativity, substitution becomes super-associative. We have

[a c [b d e]] = [a [c b d] e] = [[a c b] d e]

[a c e] = [e c a]

[a c [b d e]] = [[b d e] c a]

[a c [b d e]] = [b c [e d a]] = [e d [a c b]]

That is, variables located in odd numbered locations are multiplied, while variables in even numbered locations are multiplied as reciprocals. This is sufficiently general that, regardless of nested substitutions to any depth, a variable in an odd indexed location can be placed in any odd location, while a variable in an even indexed location can be placed in any even location.

Flat Substitution

Keeping in mind that substitution operations require three arguments (the put-, for-, and in-forms), we can construct the bracket notation reflect super-symmetry, by eliminating all inner brackets. A form such as

[a c b d e]

can represent two applications of substitution, the grouping of three sequential arguments is discretionary. This notation is particularly useful as a representation of exponentiation.

For proper form, the number of structures enclosed in brackets should be 2n+3, with n+2 structures in odd indexed (multiplication) positions, and n+1 in even indexed (division) positions. Those in even positions can be thought of as multiplying by their reciprocal. A proper application of substitution would take two structures from the even group and one structure as the for-form from the odd group. Results would be returned into the even group.

Within these repeated choosing restrictions, substitution is now also a flat function. The consideration of the order in which triplets are reduced by substitution is an important factor for the pragmatics, but not the semantics, of an implementation.

==========
Figure %%%: pick one form this bag and two from this one
==========

Sequential and Parallel Substitution

Fusion is a one-step process mapping parts to a Whole, so that the idea of applying fusion in sequential steps is not inherent within the conceptualization of the operation. In comparison, multiple substitutions are usually considered to require sequential implementations. In general,

$$[a\ c\ [b\ d\ e]] \neq [[a\ c\ b]\ d\ e]]$$

Consider this case in which e = cd, b = c, and all other letters stand for themselves:

$$[a\ c\ [c\ d\ cd]] = [a\ c\ cc] = aa$$
$$[[a\ c\ c]\ d\ cd] = [a\ d\ cd] = ca$$

Composition of substitutions is often restricted so that the for-form of one substitution and the put-form of another do not share identifiers. However, the super-symmetry that comes with the interpretation of substitution-as-division assures the ordering of substitutions does not change the result. For example,

a = ●●
b = ●
c = ●●
d = ●●●●
e = ●●●●

```
[[a b c] d e] = [[•• • ••] •••• ••••] = [•••• •••• ••••] = ••••
[a [b c d] e] = [•• [• •• ••••] ••••] = [•• •• ••••]     = ••••
[a b [c d e]] = [•• • [•• •••• ••••]] = [•• • ••]        = ••••
```

Substitution has fixed arity, it requires three arguments. Therefore applications of substitution can reduce a collection of arguments three at a time. However, nestings can be composed so that substitutions can be applied in parallel or in sequence. For the case of four applications of substitution, the nestings could be arranged to take four time steps, or three or two time steps.

```
[a b [c d [e f [g h i]]]]          four steps, one module
[a b [c [d e f][g h i]]]           three steps, two modules
[[a b c][d e f][g h i]]            two steps, three modules
```

In a resource constrained implementation, arguments may be combined in any groupings of three and in any hierarchical order. Naturally, algebraic simplification is beneficial prior to actual substitution. For example,

```
[[a c e] • [b d c]]
[ a c e  •  b d c ]
[ a • e  c  c d b ]
[ a • e       d b ]
```

Deep nesting permits recycling the same "substitution module" resource. In general, given unlimited resources, a purely sequential implementation of substitution-as-multiplication for k substitutions requires k temporal steps, while a parallel implementation provides a logarithmic reduction of temporal steps.

Normal and Applicative Order

[[a c e] d f] as do [a c e] and then subst

vs subst [a c e] for d in f.

For example,

```
[[a b c] d e] = [[•• • ••] •••• ••••] = [•••• •••• ••••] = ••••
[[a b c] d e] = [[•• • ••] •••• ••••] = [•• • ••] = ••••

[a b [c d e]] = [•• • [•• •••• ••••]] = [•• • ••]        = ••••
[a b [c d e]] = [•• • [•• •••• ••••]] = [•••• •••• ••••] = ••••

[[c d e] b a] = [[•• •••• ••••] • ••] = [•• • ••]        = ••••
```

```
      [[c d e] b a] = [[●● ●●●● ●●●●] ● ●●]
                    = [●● ●●●● ●●●●]|[●● ●●●● ●●●●] = ●●|●●   = ●●●●
```

Also have more uncommon middle-order

```
      [a [b c d] [b c d]] = [●● [● ●● ●●●●] [● ●● ●●●●]|●●] = ●●●● = ●●●●
```

```
================================================================================
            Theorems that Extend the Divide Function

sort
      I(y//x)                 W(q[● x y])
      I(rem(y/x))             W(r[● x y])
right zero
      0/x = 0                       [● x ><] = ><
left zero
      not y/0                       not [● >< y]

subtraction quotient
      if y>x then y/x = (y-x)/x + 1
                                    if ●.y|Δx then [● x y] = [● x y|Δx]|●
subtraction remainder
      if y<x then rem(y/x) = rem((y-x)/x)
                                    if ●.y|Δx then r[● x y] = r[● x y|Δx]
greater-than
      if x>y then not y//x                if ●.x|Δy then not e[● x y]
```

```
================================================================================

q[● x y] = q[● x y|Δx|x] = q[● x y|Δx]|q[● x x] = q[● x y|Δx]|●
r[● x y] = r[● x y|Δx|x] = r[● x y|Δx]|r[● x x] = r[● x y|Δx]

q[● x y|x] = q[● x y]|q[● x x] = q[● x y]|●
```

==========

Conventional Axioms for Exact Divide

exact divide

y//x = z iff x*z=y — e[• x y] = z iff [x • z] = y

e[• x y] = z iff y = [x • q[• x y]]

exact remainder

if y//x = z then rem(y/x) = 0 — e[• x y] = z iff r[• x y] = ><

addition is distribution over exact divide

y///x and z///x iff y///x and (y+z)///x

e[• x y] and e[• x z] iff e[• x y] and e[• x y|z]

multiplication

if y///x or z///x then (y*z)///x — e[• x y] or e[• x z] iff e[• x [y • z]]

e[• x y] or e[• x z] iff e[y x z]

transitivity

if y///x and z///y then z///x — e[• x y] and e[• y z] iff e[• x z]

e[e[• x y] • e[• y z]] = e[• x z]

antisymmetry not when integers

if y///x and x///y then x=y — e[• x y] and e[• y x] iff x=y

reflexivity

x///x — e[• x x] = •

gcd

gcd(x,0) = x

gcd remainder

gcd(x,y) = gcd(y, rem(x,y))

common divisor

x///gcd(x,y) and y///gcd(x,y)

greatest

if x///z and y///z then gcd(x,y)///z

==========

multiplication
if y///x or z///x then (y*z)///x e[• x y] or e[• x z] iff e[• x [y • z]]
                                  e[• x y] or e[• x z] iff e[y x z]

      [• x [y • z]]
      [• x y • z]
      [y x • • z]
      [y x z]

      e[• x y] or e[• x z] iff e[y x z]

e[y x z] iff e[• x y] or e[• x z]
e[y x z] iff not([• x y] and [• x z])
e[y x z] iff not([• x q[• x y]|r[• x y]] and [• x q[• x z]|r[• x z]])

[• x q[• x y]|r[• x y]] and [• x q[• x z]|r[• x z]]
[• x q[• x y]]|[• x r[• x y]] and [• x q[• x z]]|[• x r[• x z]]

y<x   [• x q[• x y]]|[• x r[• x y]] and [• x q[• x z]]|[• x r[• x z]]
•.x|Δy [• x   ><]|[• x   y] and [• x   ><]|[• x   z]
•.x|Δy   [• x   y] and   [• x   z]
===FINISH

addition
            e[• x y] and e[• x z] iff e[• x y] and e[• x y|z]

            e[• x y] and e[• x z] iff e[• x y] and e[• x y|z]
                                                  e[• x y]|e[• x z]
                  is just distribution over exact divide

multiplication is always exact divide
            e[a • b]

transitivity over integers
      if y///x and z///y then z///x   e[• x y] and e[• y z] iff e[• x z]
use
            e[e[• x y] • e[• y z]] = e[• x z] not --true of all

                  e[e[• x y] • e[• y z]]
                  e[e[• x y] e[• • y] z]?
                  e[e[• x y]   y z]
                  e[• e[x y   y] z]
                  e[•   x   z]

antisymmetry
      if y///x and x///y then x=y   e[• x y] and e[• y x] iff x=y

assume e[e[• x y] • e[• y x]] = •

therefore have form of [unit • unit] = •

      e[• x y] = e[• y x] = •
or
      e[• x y] = e[• y x] = ◊

for antisymmetry, don't need x=y (!)

      if y///x and x///y then x=y

            2/-2 and -2/2 but x ≠ y
===

Division of integers (including negative numbers) is not antisymmetric

Factoring of Whole Numbers Revisited

We can now examine factoring as a structural relationship between substitution forms. Factoring the number 30 will provide an example. In flat substitution notation:

            30 = 5*3*2   == [•• • ••• • •••••]

The substitution form reads "substitute 2 for each unit in 3 for each unit in 5"

==========
Figure %%%: of structural divisions of 30

[• • 30] = [•• • 15] = [••• • 10] = [••••• • ••••••]
= [[•• • •••] • •••••] = [•• • [••• • •••••]] = [•• [• ••• •] •••••]

(15|15) = (10|10|10) = (5|5|5|5|5|5)
       = (3|3|3|3|3|3|3|3|3|3) = (2|...|2) = (1|...|1)
==========

Factors are computed via partitioning. We begin with the ensemble of 30 units, and subdivide into equal parts. Alternatively we could begin with the maximal partition of 30 into unit wholes, and remove partitions symmetrically by exact

division. Since ensembles are spatial forms, the determination of factors can be done by spatial and well as symbolic symmetries.

==========
Figure %%%: spatial division of 30
==========

division decomposition

[a c e] = [a [c • n][e • n]]

THEORY OF NONNEGATIVE INTEGERS

Constant:
```
0
```
Constructor Function:
```
x+
```
Relation:
```
Integer[x]
```
Mixin equality:
```
=
```

Terms:
```
0, 0+, 0++, 0+++. ...
```

To shorten the notation, we count the number of `+`'s and call the term that number, to get `0, 1, 2, ...` (This is an example of swapping computation for definition.)

Axioms:

Generate base: `Integer[0]`

Generate successor: `Integer[x+]`

Unique zero: `not (x+ = 0)`

Unique successor: `(x+ = y+)  ->  (x = y)`

Computational Technique (Induction) (Loop):
```
S[0] and (S[x] -> S[x+])  ->  S[x]
```

Proof and Computation are defined by showing a base case, `S[0] == true`, and by assuming a general case, `S[x]`, and then showing the next case, `S[x+] == true`. This works because we know we can iterate over all numbers from `0` to `x` using only the stepping function `+`.

In recursive programming we do exactly the same thing (but usually in reverse, using a inverse theory that has Predecessor x- rather than Successor x+. For example, to sum, we exit at a base case, and iterate over each of the others explicitly:

```
Sum[x]  ==  if (x = 0) then 0 else x + Sum[x-]
```

The computational technique can also be recognized as iteration:

```
Sum[x]  ==  loop from 0 to x doing  (result := result + x)
```

Now I know that the machine does all this stuff reliably and it seems silly to spend so much time on it, *but* the conventional machine requires instances, it needs bound variables to work. The above

techniques work on symbols, variables, unknowns, as well as on values, instances, knowns. Besides, it's not what the machine does that matters to us, it's what we understand about what the machine does from looking at i/o. If you're serious about low level objects, here's how integer objects manage to count. They don't flip bits, they pass "Add yourself to this" messages. #

Computational techniques:
Substitution in general
Decomposition: (x = 0) or (x = y+)

The Decomposition rule lets us define x- for all x except 0. It does this by introducing an arbitrary y.

Now we can expand to include addition and multiplication: #

Axioms for Addition:
Addition base: (x + 0) = x
Addition loop: (x + y+) = (x + y)+

Rules for Addition:
Integer[(x + y)]
(x + 1) = x+

Computational techniques:
Substitution
(x + z) = (y + z) == (x = y)
(x + y) = 0 == (x = 0) and (y = 0)

Axioms for Multiplication:
Multiplication base: (x * 0) = 0
Multiplication loop: (x * y+) = (x * y) + x

Rules for Multiplication:
(x * 1) = x

When x and y are different structures (the theory is applied to different domains), + and * have different definitions. The implementation changes (overloading) but the abstract organization stays the same.

I'm starting to abbreviate here, omitting the things that get inherited from other theories, and that are very much like theories previously described. Don't forget that there are an infinite number of rules, and substitution creates an infinite number of expressions, so let's just concentrate on the essentials. #

Mixin Associative Ordering:

```
<,  =,  ≤
```

Mix-in means build a world with both theories, explicitly coupling the theories with mixed rules. The world is not organizationally different because the theories maintain their unique organization while working in concert. This is how to achieve modularity of object-oriented systems. Coupling rules specify communication protocols.

Structural extensions:

```
Positive[x]  ==  Integer[x] and not (x = 0)
Minimum[x,y]  ==  (if (x <= y) then y else x)
```

Predecessor: `(x + 1)- = x`
Subtraction base: `(x - 0) = x`
Subtraction loop: `(x+ - y+) = (x - y)`
`(x + y) - y = x`

Quotient base: `(x < y)  ->  (x/y = 0)`
Quotient loop: `(x + y)/y = x/y + 1`

Remainder base: `(x < y)  ->  (Remains[x,y] = x)`
Remainder loop: `Remains[(x + y), y] = Remains[x,y]`

Quotient-Remainder: `(x = (y*(x/y)) + Remains[x,y]) and (Remains[x,y] < y)`

Divides: `(x D y) == (x * z = y) and Integer[z]`
`(x D y) == (Remains[y,x] = 0)`
Multiply-divide: `(x D y) or (x D z)  ->  (x D y*z)`

GreatestCommonDivisor Base: `Gcd[x,0] = x`
GreatestCommonDivisor Loop: `(y = 0) or Gcd[x,y] = Gcd[y, Remains[x,y]]`

```
(Gcd[x,y] D x)  and  (Gcd[x,y] D y)
(z D x) and (z D y)  ->  (z D Gcd[x,y])
```

In Divides, Integer takes care of the existence of the z, cause

```
Integer[non-existing integer] == false
```

This could be written as

```
(x D y) == (x * Ez = y)
```

UNIT AND BOUNDARY ARITHMETICS -- SECTION IVE October 1, 2008 10:55 PM
William Bricken
January 2007

STRUCTURE OF UNIT EXPONENTS

In substitution-as-multiplication, the square of a number a is a self-substitution

a^2 == [a • a]

Higher powers are explicitly represented as a sequence of self-substitutions:

a^3 == [a • [a • a]]

a^4 can be represented in both the "repeated substitution" format and in the hierarchical substitution format:

a^4 = a*a*a*a == [a • [a • [a • a]]]

a^4 = (a*a)^2 == [[a • a] • [a • a]]

The first form corresponds to sequential binary multiplication, the second to parallel binary multiplication.

Both versions require three substitutions, however two of the hierarchical multiplications can be achieved at the same time. Thus the sequential version requires on "multiplication module" and three time steps, while the parallel version requires two multiplication modules and two time steps, a classic time/space tradeoff.

For exponentiation, the flat substitution notation is more convenient, since exponentiation substitution structures show great regularity.

Substitution-as-multiplication does not have an explicit notation for exponentiation, so we will provide one. The right superscript conventionally used to denote exponentiation is a shorthand that does not represent what it stands for. Similarly our convenience notation does not directly represent the substitution structure of iterated (alternatively hierarchical) multiplications. However, powers of unit ensembles do have a distinct substitution structure, as is shown in Table %%%, and it is this structure that we will abbreviate.

a^b == [a • b..]
a^-b == [• a b..]

The encoded token b.. stands in place of the power b in a^b. Using this notation, the example cases are shown in Figure %%%

==========

Summary of Substitution Forms for Exponents

```
a^-b  ==                          [• a b..]
a^-3  ==    [• a • a • a •]   =   [• a •••..]
a^-2  ==      [• a • a •]     =   [• a ••..]
a^-1  ==        [• a •]       =   [• a •..]
a^0   ==        [• a a]       =   [• a ..]
a^1   ==        [a • •]       =   [a • •..]
a^2   ==        [a • a]       =   [a • ••..]
a^3   ==      [a • a • a]     =   [a • •••..]
a^4   ==    [a • a • a • a]   =   [a • ••••..]
a^b   ==                          [a • b..]
```

==========

The notation is awkward for unit ensembles, since we have not yet granted numeral abbreviations for ensembles. It is not intended to be useful for computation nor is it succinct. We need the notation solely to conveniently express various axioms about exponents within the unit ensemble system.

Powers can be represented as substitutions in a variety of ways, each variety suggests an implementation strategy. For example

a^4 == [a • a • a • a] = [a • [a • [a • a]]] = [a [• [a • a] •] a]

a^-3 == [• a • a • a •] = [• a [• a [• a •]]] = [• [a [• a •] a] •]

===
log

```
a^4          ==    [a • a • a • a]   =  [a • ••••..]

                       log [a • a • a • a] = ••••
loga a^4 = 4           log [a • ••••..] = ••••..
```

===
SHOW

```
x^a • x^b = x^(a+b)
x^-a = 1/x^a
(x^a)^b - x^(a*b)
x^a * y^a = (x*y)^a
x^0 = 1
[[x • a..] • [x • b..]] = x • a|b..]
[• x a..] = [• [x• a..] •]
[x x •] = •
[[x • a..] • [y • a..]] = [[x • y] • a..]
```

==========

Unit Ensemble Axioms of Whole Number Arithmetic Exponentiation

for all integer x,y,z:

exponent zero		
	x^0 = 1	[• x x] = [• x ..] = •
exponent successor		
	x^(y+1) = (x^y)*x	[x • y|•..] = [x • [x • y..]]

Properties

sort		
	I(x^y)	W([x • y..])
exponent one		
	x^1 = x	[x • •] = [x • •..] = x
base zero		
	if y ≠ 0 then 0^y=0	[>< • y..] = ><
exponent plus		
	x^(y+z) = (x^y)*(x^z)	[x • y|z..] = [[x • y..] • [x • z..]]
exponent times		
	x^(y*z) = (x^y)^z	[x • [y • z]..] = [[x • y..] • z..]

==========

Comparison of Axioms

The zero and one rules reduce to simple substitution structures, although they can be written in the exponentiation notation. Consistent with the intention of unit ensemble arithmetic, it is preferred not to introduce new notation for known forms, since having different syntactic forms of the equivalent substitutions only adds complexity that must then be reduced via transformation.

Exponent successor is an applied case of exponent plus, and therefore unnecessary. What is left of interest is only the distribution relations between exponentiation and both fusion and substitution.

In exponent plus, fusion of a power converts to substitution of the fused powers. This rule asserts that the sum of exponents is equal to the product of the bases. In exponent times, the substitution structure as a power converts to substitution of one put/in-form into the other.

Pure distribution of ".." over substitution helps to clarify the rules for nested exponents:

a^b^c as either a^(b^c) or as (a^b)^c

a^(b^c) [a • [b • c..]..]
(a^b)^c [[a • b..] • c..]

(a^b)^c = a^(b*c) [[a • b..] • c..] = [a • [b • c]..]

=====

Examples

(a+b)^2 + (a-b)^2 = 2(a^2 + b^2)

a^-2 == [• [a • a] •]

(-a)^2 == [[• ◊ a] • [• ◊ a]]

==DO

```
  [[• ◊ a] • [• ◊ a]]
= [[• ◊ a] [• • ◊] a]
= [[• ◊ a] [◊ • •] a]
= [[• ◊ a]      ◊  a]
```

```
= [a [◊ • •] [a ◊ •]]
= [a    ◊    [a ◊ •]]
= [a    ◊    [• ◊ a]]
= [a   [◊     • ◊] a]
= [a          •    a]              ==  a^2
```

(a + b)^2 == [a|b • a|b]

```
  [a|b • a|b]
= [a • a]|[a • b]|[a • b]|[b • b]
= [a • a]|[[a • b] • ••]|[b • b]   ==  (a^2 + 2ab + b^2)
```

(a + b)(a - b) == [a|b • a|Δb]

```
  [a|b • a|Δb]
= [a • a]|[a • Δb]|[b • a]|[b • Δb]
= [a • a]|Δ[a • b]|[b • a]|Δ[b • b]
= [a • a]|                Δ[b • b]
```

= [a • a]|Δ[b • b] == a^2 - b^2

(x^2 + 3x - 4) = (x + 4)(x - 1)

[x • x]|[•••• • x]|•••• =

[x|•••• • x|◊]
= [x • x]|[◊ • x]|[•••• • x]|[•••• • •]
= [x • x]|[◊|•••• • x]|••••
= [x • x]|[••• • x]|••••

(a^2 + b^2)(c^2 + d^2) = (ad + bc)^2 + (bd - ac)^2

[[a • a]|[b • b] • [c • c]|[d • d]]

[[a • a] • [c • c]]|[[a • a] • [d • d]]|[[b • b] • [c • c]]|[[b • b] • [d • d]]
[[a • c] • [a • c]]|[[a • d] • [a • d]]|[[b • c] • [b • c]]|[[b • d] • [b • d]]

[[a • d]|[b • c] • [a • d]|[b • c]]|Δ[[b • d]|[a • c] • [b • d]|[a • c]]

=====

Logarithms

log of a boundary form counts the number of nested parens

2^5 = 32 (•)^((•))• = (((((•)))))

log2 32 = 5 log2 (((((•))))) = ((•))•

Roots

The unit ensemble calculus requires discrete units, and thus cannot express irrational numbers. (We have developed a boundary mathematics that addresses the full range of reals, the James calculus [ref].) However, we should be able to determine the two square roots of square numbers. That procedure is:

STRUCTURE OF FRACTIONS

In conventional notation, fractions consist of two numbers. Division of the two is not performed. We might thus expect unit ensemble fractions not to resolve into a single Whole. Consider the simplest case of the fraction 1/2:

1/2 == [• •• •]

This illustrates another case of failure-to-match. In general, when c>e,

[• c e]

will not match, because there are more unit sin c than are in e. In these cases, the substitution does not proceed.

This substitution could be considered to be implicit, the unresolved substitution operation itself representing the fraction. (There are methods for constructing the division analogous to that of decimals that we will not discuss here.)

In general,

```
  1/c   ==  [• c •] = [◊ c ◊]
  a/c   ==  [a c •] = [• c a]
-(a/c)  ==  [a c ◊] = [◊ c a]
```

The axioms of rational number include the rules of adding and multiplying fractions. We first consider the simpler case of equivalent fractions.

Equivalent Fractions

Equivalent fractions are those that share a divisor across both numerator and denominator. For example:

1/2 = 2/4 = 3/6 = ...

[• •• •] = [• •••• ••] = [• •••••• •••] = ...

In general,

[• c e] = [• [c • n] [e • n]] = [• [c n •] [e n •]]

```
  [• [c • n] [e • n]]
= [• [c • n] [n • e]]
= [[• c •] [n n •] e]
```

```
                = [[• c •]      •  e]
                = [• [c •      •]  e]
                = [•  c           e]
```

This relationship can also be illustrated using partitions:

```
            [• (•|•) •] = [• (••|••) ••] = [• (•••|•••) •••] = ...
```

===
finish
===

Multiplying Fractions

The transformation rule governing multiplication of fractions is:

```
            a/c * b/d = a*b / c*d
```

Transcribing into unit ensemble notation:

```
            a/c  ==  [• c a]
            b/d  ==  [• d b]
            a*b  ==  [a • b]
            c*d  ==  [c • d]

      a/c * b/d  ==  [[• c a] • [• d b]]

      a*b / c*d  ==  [a [c • d] b]
```

The left-hand-side is converted into the right-hand-side using substitution rules:

```
        [[• c a] • [• d b]]
      = [[a c •] • [• d b]]
      = [ a [c • •][• d b]]
      = [ a  c     [• d b]]
      = [ a [c      • d] b]
```

Dividing Fractions

Conventionally, division is multiplication by the reciprocal. In substitution-as-division, the notion of reciprocal is achieved by exchanging the put-form with the for-form. Division by fractions follows the same pattern. For example,

```
1/3 ÷ 1/2 = 1/3 * 2/1 = 2/3

[• ••• •]/[• •• •] = [• ••• •]*[• • ••]
                   = [• ••• [• • ••]]
                   = [• [••• • •] ••]
                   = [•  •••     ••]
```

Substitution-as-division permits direct expression of multiplication by a reciprocal:

```
[• ••• •]/[• •• •] = [[• ••• •] • [• [• •• •] •]]
                   = [[• ••• •] • [[• • ••] • •]]
                   = [[• ••• •] • [    ••  • •]]
                   = [[• ••• •] •      ••     ]
                   = [• [••• • •]      ••     ]
                   = [•  •••           ••     ]
```

Division can also be achieved by direct substitution into the for-form rather than the in-form:

```
[• ••• •]/[• •• •] = [• [••• • •] •]/[• •• •]
                   = [• [••• [• •• •] •] •]
                   = [• [••• • [•• • •]] •]
                   = [• [••• •  ••    ] •]
```

==========

Figure %%%:

```
2/3 ÷ 5/6 = 2/3 * 6/5 =  12/15 = 4/5

[•• ••• •]/[••••• •••••• •] = [[•• ••• •] [••••• •••••• •] •]

       [[•• ••• •] [••••• •••••• •]  •]
  =    [[•• ••• •]  ••••• [•••••• • •]]
  =    [[•• ••• •]  •••••  ••••••    ]
  =    [•• ••• [•  •••••  ••••••]    ]
  =    [•• ••• [••••••  •••••  •]    ]
  =    [[•• ••• ••••••]  •••••  •    ]
  =    [[•• ••• •••|•••] •••••  •    ]
  =    [         ••|••   •••••  •    ]
  =    [          ••••   •••••  •    ]
```

==========

For readability, we can replace each unit ensemble with the numeral that names it. The substitution semantics remains the same:

```
================================================================================
Figure %%%:
            [2 3 •]/[5 6 •] = [[2 3 •][5 6 •] •]

                    [[2 3 •][5 6 •]  •]
                =   [[2 3 •] 5 [6 • •]]
                =   [[2 3 •] 5  6     ]
                =   [2 3 [•  5  6]    ]
                =   [2 3 [6  5  •]    ]
                =   [[2 3 6] 5  •     ]
                =   [[2 3 3|3] 5  •   ]
                =   [     2|2  5  •   ]
                =   [      4   5  •   ]
================================================================================
```

In general, by the method of division as multiplication by a reciprocal:

```
            a/c ÷ b/d = a/c * d/b = ad/cb

            [a c •]/[b d •] = [a c •]*[d b •]
                            = [a c [d b •]]     ==fix==
                            = [a c [• b d]]
                            = [a [c • b] d]
```

We can also formulate division directly as substitutions, without relying on a reciprocal:

```
            x ÷ y  == [x y •]   with x = [a c •],  y = [b d •]

            a/c ÷ b/d  ==    [[a c •] [b d •] •]
                           = [[a c •] b [d • •]]
                           = [[a c •] b  d     ]
                           = [a [c • b] d]
```

Adding and Subtracting Fractions

Substitution is blind to unit type, treating Solid and Hollow units as the same. We consider next addition of fractions with like and with unlike denominators

Like Denominators

Adding fractions with like denominators is a straightforward application of distribution of fusion over substitution:

a/c + b/c = (a + b)/c

Transcribing:

a/c == [• c a]
b/c == [• c b]

[• c a]|[• c b] = [• c a|b]

Unlike Denominators

For addition of fractions, we will call upon distribution of fusion over substitution:

a/c + b/d = (a*d + b*c)/c*d

Transcribing:

a/c == [• c a]
b/d == [• d b]
a*d == [a • d]
b*c == [b • c]
c*d == [c • d]

a/c + b/d == [• c a]|[• d b]

(a*d + b*c)/ c*d == [• [c • d] [a • d]|[b • c]]

To apply fusion, as well as having the same for-form, the fused substitutions require either the put-form or the in-form to be the same. We can construct this as follows: To convert unit ensemble fractions to a common denominator, we construct a common for-form in both fused substitutions. This permits distribution of substitution over fusion.

[• c a]
= [• [c • •] a]
= [• [c [• d d] •] a]
= [• [[c • d] d •] a]
= [• [c • d][d • a]]
= [• [c • d][a • d]]

Each step of the above transformation is either an application of global substitution, or a rearrangement of triplets. Using super-associativity, this can be more simply expressed as:

```
  [•           c a ]
= [•  • • d  d c a ]
= [•  c • d  d • a ]
= [• [c • d][d • a]]
```

Continuing for the other added fraction,

```
  [•   d            b ]
= [• [ d  •       •] b ]
= [• [ d [• c  c] •] b ]
= [• [[d  • c] c  •] b ]
= [•  [d  • c][c  •  b]]
= [•  [d  • c][b  •  c]]
= [•  [c  • d][b  •  c]]
```

Fusing these,

```
              [• c a]     |       [• d b]
     = [• [c • d] [a • d]]|[• [c • d] [b • c]]
     = [• [c • d] [a • d] |[b • c]           ]
```

The requirement of "like denominators" for addition of fractions is reflected in substitution-as-division as the requirement of two identical substitution forms to achieve fusion.

```
           [a c e]|[a c f] = [a c e|f]
```

==========

Figure %%%: Axioms of Rational Numbers

```
n/1 = n                        [• • n] = n

0/n = 0   n ≠ 0   W[n]         [• n ><] = ><

n/-a = -n/a                    [• Δa n] = [• a Δn]
```

equivalent fraction reduction

```
     x/y = (x/gcd(x,y)) / (y/gcd(x,y))     gcd(x,y) ≠ 1

     x/y = x/gcd(x,y) * gcd(x,y)/y     gcd(x,y) ≠ 1
```

==========

Unit Ensemble Algebra

We now consider algebraic manipulations in unit arithmetic. Although transformation rules can be formulated over equations, here we focus on the decomposition as an algebraic technique,

a = b iff a|n = b|n

Combining Like Terms

Combining like algebraic terms calls upon conventional distribution of multiplication over addition. For ensembles, it is distribution of substitution over fusion. In general,

ax + bx = (a + b)x

[a • x]|[b • x] = [a|b • x]

Solving Linear Equations

Additive

x + 3 = 5

x|●●● = ●●●●●
x|●●● = ●●|●●●
x = ●●

A void-based method for solving this equation follows:

x|●●●|Δ●●●●● = ><

x●●●◊◊◊◊◊
x◊◊
x|Δ◊◊ = ><
x = ΔΔ◊◊ = ●●

Multiplicative

use x=y iff [a c x]=[a c y]

2x = 6

```
[●● ● x] = ●●●●●●
[●● ● x] = [● ● ●●●●●●]
[●● ● x] = [● ● ●●●|●●●]
[●● ● x] = [●● ● ●●●]
x = ●●●
```

The equivalent void-based method:

```
[●● ● x]|Δ●●●●●● = ><
[●● ● x]|Δ[● ● ●●●●●●] = ><
[●● ● x]|Δ[● ● ●●●|●●●] = ><
[●● ● x]|Δ[●● ● ●●●] = ><
[●● ● x]|[●● ● Δ●●●] = ><
[●● ● x]|[●● ● ◊◊◊] = ><
[●● ● x|◊◊◊] = ><
x|◊◊◊ = ><
x|Δ●●● = ><
x = ●●●
```

Simple Linear Equation

```
2x + 3 = 7
```

```
[●● ● x]|●●● = ●●●●●●●
[●● ● x]|●●● = ●●●●|●●●
[●● ● x] = ●●●●
[●● ● x] = [● ● ●●●●]
[●● ● x] = [● ● ●●|●●]
[●● ● x] = [●● ● ●●]
x = ●●
```

void-based:

```
[●● ● x]|●●● = ●●●●●●●
[●● ● x]|●●●|Δ●●●●●●● = ><
[●● ● x]|●●●◊◊◊◊◊◊◊   = ><
[●● ● x]|◊◊◊◊         = ><
[●● ● x]|[● ● ◊◊◊◊]   = ><
[●● ● x]|[●● ● ◊◊]   = ><
[●● ● x|◊◊]   = ><
x|◊◊   = ><
x|Δ●●  = ><
x = ●●
```

Solving Proportions

6/9 = 2/x

[●●●●●● ●●●●●●●●● ●] = [●● x ●]
[●●|●●|●● ●●●|●●●|●●● ●] = [●● x ●]
[[●● ● ●●●] [●●● ● ●●●] ●] = [●● x ●]
[●● [● ●●● ●●●] [● ●●● ●]] = [●● x ●]
[●● ● [● ●●● ●]] = [●● x ●]
[●● [● ● ●●●] ●] = [●● x ●]
[●● ●●● ●] = [●● x ●]
[●● ●●● ●] = [●● x ●]
x = ●●●

Cross Multiply

[a c ●]=[b d ●] iff [a ● d]=[b ● c]

Quadratic Equation

Completing the Square

[x ● x]|[a ● x] = [x|[a ●● ●] ● x|[a ●● ●]]|[[a ●● ●] ● [a ●● ●]]

Pythagorean Theorem

[a ● a]|[b ● b] = [c ● c]

[c|a b ●] = [b c|Δa ●]

cross-mult:

[c|a ● c|Δa] = [b ● b]

[c|a ● c|Δa] = [c ● c]|Δ[a ● a]

Systems of Equations

x+y = 1
x-y = 7
not intended for matrix techniques

Already done as algebraic fusion and substitution rules.

```
================================================================================
Figure %%%:  Integer Sum and Factorial

      ∑1 = 1                          ∑● = ●
      ∑x = x + ∑(x-1)                 ∑x = x|∑.x|◊
      ∑(x+1) = (x+1) + ∑x             ∑.x|● = x|●|∑x

zero
      0! = 1                          ●! = ●
successor
      x! = x*(x-1)!                   x! = [x ● x|◊.!]
      (x+1)! = (x+1)*x!               x|●.! = [x|● ● x!] = [x ● x!]|x!

================================================================================
```

STRUCTURE OF UNIT EXPONENTS -REDO WITH {} 5/3/07

In substitution-as-multiplication, the square of a number a is a self-substitution

a^2 == [a • a]

Higher powers are explicitly represented as a sequence of self-substitutions:

a^3 == [a • [a • a]]

a^4 can be represented in both the "repeated substitution" format and in the hierarchical substitution format:

a^4 = a*a*a*a == [a • [a • [a • a]]]

a^4 = (a*a)^2 == [[a • a] • [a • a]]

The first form corresponds to sequential binary multiplication, the second to parallel binary multiplication.

Both versions require three substitutions, however two of the hierarchical multiplications can be achieved at the same time. Thus the sequential version requires on "multiplication module" and three time steps, while the parallel version requires two multiplication modules and two time steps, a classic time/space tradeoff.

For exponentiation, the flat substitution notation is more convenient, since exponentiation substitution structures show great regularity.

Substitution-as-multiplication does not have an explicit notation for exponentiation, so we will provide one. The right superscript conventionally used to denote exponentiation is a shorthand that does not represent what it stands for. Similarly our convenience notation does not directly represent the substitution structure of iterated (alternatively hierarchical) multiplications. However, powers of unit ensembles do have a distinct substitution structure, as is shown in Table %%%, and it is this structure that we will abbreviate. Consistent with the spatial approach, the abbreviation is a boundary, a new boundary {}.

a^b == {a • b}
a^-b == {• a b}

The encoded boundary {.. .. b} stands in place of the power b.

Importantly, commutativity of substitution does not hold with "power boundaries".

Using this notation, the example cases are shown in Figure %%%

```
================================================================================
Summary of Substitution Forms for Exponents

            a^-b  ==                          {• a b}
            a^-3  ==    [• a • a • a •]   =   {• a ●●●}
            a^-2  ==      [• a • a •]     =   {• a ●●}
            a^-1  ==        [• a •]       =   {• a ●}
            a^0   ==        [• a a]       =   {• a ><}
            a^1   ==        [a • •]       =   {a • ●}
            a^2   ==        [a • a]       =   {a • ●●}
            a^3   ==      [a • a • a]     =   {a • ●●●}
            a^4   ==    [a • a • a • a]   =   {a • ●●●●}
            a^b   ==                          {a • b}
================================================================================
```

The notation is awkward for unit ensembles, since we have not yet granted numeral abbreviations for ensembles. It is not intended to be useful for computation nor is it succinct. We need the notation solely to conveniently express various axioms about exponents within the unit ensemble system.

The convenience of the power boundary is that all substitution transformation rules other than commutativity hold.

Powers can be represented as substitutions in a variety of ways, each variety suggests an implementation strategy. For example

```
      a^4   ==  [a • a • a • a] = [a • [a • [a • a]]] = [a [• [a • a] •] a]

      a^-3  ==  [• a • a • a •] = [• a [• a [• a •]]] = [• [a [• a •] a] •]
```

===
log

```
            a^4           ==    [a • a • a • a]   =  {a • ●●●●}

                                log [a • a • a • a] = ●●●●
            loga a^4 = 4        log {a • ●●●●} = ●●●●
```

===
SHOW

x^a • x^b = x^(a+b)
x^-a = 1/x^a
(x^a)^b - x^(a*b)
x^a * y^a = (x*y)^a
x^0 = 1
[{x • a} • {x • b}] = {x • a|b}
{• x a} = [• {x • a} •]
[x x •] = •
[{x • a} • {y • a}] = {[x • y] • a}

last is distribution of exponential over multiplication/substitution

==========

Unit Ensemble Axioms of Whole Number Arithmetic Exponentiation

for all integer x,y,z:

exponent zero		
	x^0 = 1	[• x x] = {• x ><} = •
	x^0 = 1	{x • ><} = {• x ><} = •
exponent successor		
	x^(y+1) = (x^y)*x	{x • y|•} = [x • {x • y}]
	x^(y+1) = (x^y)*x	{x • y|•} = {{x • y} • {x • •}}
or reinterpret | inside {} as • NO!		
	x^(y+1) = (x^y)*x	{x • y|•} = {{x • y} | {x • •}}

Properties

sort		
	I(x^y)	W({x • y})
exponent one		
	x^1 = x	[x • •] = {x • •} = x
base zero		
	if y ≠ 0 then 0^y=0	{>< • y} = ><
exponent plus		
	x^(y+z) = (x^y)*(x^z)	{x • y|z} = [{x • y} • {x • z}]
exponent times		
	x^(y*z) = (x^y)^z	{x • [y • z]} = {{x • y} • z}

==========

Comparison of Axioms

The zero and one rules reduce to simple substitution structures, although they can be written in the exponentiation notation. Consistent with the intention of unit ensemble arithmetic, it is preferred not to introduce new notation for known forms, since having different syntactic forms of the equivalent substitutions only adds complexity that must then be reduced via transformation.

Exponent successor is an applied case of exponent plus, and therefore unnecessary. What is left of interest is only the distribution relations between exponentiation and both fusion and substitution.

In exponent plus, fusion of a power converts to substitution of the fused powers. This rule asserts that the sum of exponents is equal to the product of

the bases. In exponent times, the substitution structure as a power converts to substitution of one put/in-form into the other.

Pure distribution of {} over substitution helps to clarify the rules for nested exponents:

a^b^c as either a^(b^c) or as (a^b)^c

a^(b^c) {a • {b • c}}
(a^b)^c {{a • b} • c}

(a^b)^c = a^(b*c) {{a • b} • c} = {a • [b • c]}

=====

Examples

(a+b)^2 + (a-b)^2 = 2(a^2 + b^2)

a^-2 == [• [a • a] •]

(-a)^2 == [[• ◊ a] • [• ◊ a]]

==DO

[[• ◊ a] • [• ◊ a]]
= [[• ◊ a] [• • ◊] a]
= [[• ◊ a] [◊ • •] a]
= [[• ◊ a] ◊ a]

= [a [◊ • •] [a ◊ •]]
= [a ◊ [a ◊ •]]
= [a ◊ [• ◊ a]]
= [a [◊ • ◊] a]
= [a • a] == a^2

(a + b)^2 == [a|b • a|b]

[a|b • a|b]
= [a • a]|[a • b]|[a • b]|[b • b]
= [a • a]|[[a • b] • ••]|[b • b] == (a^2 + 2ab + b^2)

(a + b)(a - b) == [a|b • a|Δb]

```
  [a|b • a|Δb]
= [a • a]|[a • Δb]|[b • a]|[b • Δb]
= [a • a]|Δ[a • b]|[b • a]|Δ[b • b]
= [a • a]|                Δ[b • b]
= [a • a]|Δ[b • b]                ==  a^2 - b^2
```

(x^2 + 3x - 4) = (x + 4)(x - 1)

```
[x • x]|[•••• • x]|•••• =

  [x|•••• • x|◊]
= [x • x]|[◊ • x]|[•••• • x]|[•••• • •]
= [x • x]|[◊|•••• • x]|••••
= [x • x]|[••• • x]|••••
```

(a^2 + b^2)(c^2 + d^2) = (ad + bc)^2 + (bd - ac)^2

[[a • a]|[b • b] • [c • c]|[d • d]]

[[a • a] • [c • c]]|[[a • a] • [d • d]]|[[b • b] • [c • c]]|[[b • b] • [d • d]]
[[a • c] • [a • c]]|[[a • d] • [a • d]]|[[b • c] • [b • c]]|[[b • d] • [b • d]]

[[a • d]|[b • c] • [a • d]|[b • c]]|Δ[[b • d]|[a • c] • [b • d]|[a • c]]

=====

Logarithms

log of a boundary form counts the number of nested parens

2^5 = 32 (•)^((•))• = (((((•)))))

log2 32 = 5 log2 (((((•))))) = ((•))•

Roots

The unit ensemble calculus requires discrete units, and thus cannot express irrational numbers. (We have developed a boundary mathematics that addresses

the full range of reals, the James calculus [ref].) However, we should be able to determine the two square roots of square numbers. That procedure is:

WORK ON quadratic formula

ax^2 + bx + c = 0

x^2 + (b/a)x + (c/a) = 0

x^2 + (b/a)x + (b^2/4a^2) + (c/a) - (b^2/4a^2) = 0

(x + b/2a)^2 + (4ac - b^2)/4a^2 = 0

(x + b/2a)^2 = (b^2 - 4ac)/4a^2
below

(x + b/2a) = +/- ((b^2 - 4ac)/4a^2)^(1/2)
x = -b/2a +/- ((b^2 - 4ac)/4a^2)^(1/2)
x = (-b +/- (b^2 - 4ac)^(1/2))/2a

above
(x + b/2a)^2 = (b^2 - 4ac)/4a^2

(x + b/2a)^2 = (b/2a)^2 - c/a

((2ax + b)/2a)^2 = (b/2a)^2 - c/a

(2ax + b)^2/4a^2 = (b/2a)^2 - c/a

(2ax + b)^2 = 4a^2((b/2a)^2 - c/a)

(2ax + b)^2 = 4a^2(b^2/4a^2 - c/a)

(2ax + b)^2 = b^2 - 4a^2 c/a

(2ax + b)^2 = b^2 - 4ac

(2ax + b) (2ax + b) = b^2 - 4ac

(2ax)^2 + 2(2ax)b + b^2 = b^2 - 4ac

(2ax)^2 + 4abx + b^2 = b^2 - 4ac

(2ax)^2 + 4abx + 4ac = 0

```
4a^2x^2 + 4abx + 4ac = 0

ax^2 + bx + c = 0

REdo above backwards

axx + bx + c = 0

4a (axx + bx + c) = 0

4aaxx + 4abx + 4ac = 0

(2ax)(2ax) + 2b(2ax) + 4ac = 0

(2ax)(2ax) + 2b(2ax) + bb - bb + 4ac = 0

(2ax + b)(2ax + b) - bb + 4ac = 0

(2ax + b)^2 = b^2 - 4ac

(2ax + b) = +/- (b^2 - 4ac)^(1/2)

[a • x • x]|[b • x]|c  =  void
[4 • a • [a • x • x]|[b • x]|c] =  void
[4 • a • a • x • x] | [4 • a • b • x] | [4 • a • c] =  void

[2 • 2 • a • a • x • x] | [2 • 2 • a • b • x] | [4 • a • c] =  void

[[2 • a • x] • [2 • a • x]] | [[2 • a • x] • 2 • b] | [4 • a • c] =  void

let D = [2 • a • x]

[D • D] | [D • b] | [D • b] | [4 • a • c] =  void

[D • D] | [D • b] | [D • b] | [b • b] | Δ[b • b] | [4 • a • c] =  void

[D • D|b] | [D|b • b] | Δ[b • b] | [4 • a • c] =  void

[D|b • D|b] | Δ[b • b] | [4 • a • c] =  void

[D|b • 2..] | Δ[b • b] | [4 • a • c] =  void

use 2.. as {}

{D|b • 2} | Δ{b • 2} | [4 • a • c] =  void
```

{D|b • 2} = {b • 2} | Δ[4 • a • c]

{{D|b • 2} 2 •} = { {b • 2}|Δ[4 • a • c] 2 •}

{D|b {• 2 2} •} = { {b • 2}|Δ[4 • a • c] 2 •}

{D|b • •} = { {b • 2}|Δ[4 • a • c] 2 •}

D|b = { {b • 2}|Δ[4 • a • c] 2 •}

D = Δb | { {b • 2}|Δ[4 • a • c] 2 •}

[2 • a • x] = Δb | { {b • 2}|Δ[4 • a • c] 2 •}

x = [• 2 • a Δb|{ {b • 2}|Δ[4 • a • c] 2 •}]

```
[a • x • x]|[b • x]|c  =  void

[• a [a • x • x]|[b • x]|c ]

[• a a • x • x]|[• a b • x]|[• a c]

[• • x • x]|[b a • • x]|[• a c]

[x • x]|[b a x]|[• a c]
[x • x]|[x a b]|[• a c]

[x • x]|[x a b]|[• 4 • a b a b]  |  [• a c]|Δ[• 4 • a b a b]
[x • x]|[x a b]|[• 4 • a b a b]  |  [• a c]|Δ[• a • 4 b a b]
[x • x]|[x a b]|[• 4 • a b a b]  |  [• a c|Δ[• 4 b a b]]

[x • x]|[• • b a x]|[b 4 b a • a •]
[x • x]|[x • b a •]|[b 4 b a • a •]
[x • x]|[2 2 x • b • • a •]|[b 2 • 2 b a • a •]
[x • x]|[2 • x • b 2 • a •]|[b 2 • 2 b a • a •]
[x • x]|[x • b 2 • a •]|[x • b 2 • a •]|[b 2 • a b 2 • a •]
[x • x]|[x • b 2 • a •]|[x • b 2 • a •]|[b 2 • a • • b 2 • a •]


[ x • x|[• 2 • a b] ]|[b 2 • a x|[• 2 • a b] ]
[ x • x|[• 2 • a b] ]|[b 2 • a • • x|[• 2 • a b] ]


[ x|[• 2 • a b] • x|[• 2 • a b] ] |[• a c]|Δ[b 4 b a • a •]
```

UNIT AND BOUNDARY ARITHMETICS -- SECTION V   October 1, 2008 10:55 PM
William Bricken
January 2007

==section 2 has base-1, now base-2, base-10

BOUNDARY ARITHMETIC AND DEPTH-VALUE NOTATION
Place value as Syntactic Sugar
Unit Ensembles Expressed in Place-value Notation
Place-value Fusion
Place-value Substitution

Base-10 Notation
Canonical Form of Numbers
READING BOUNDARY INTEGERS
? INSIDE AND OUTSIDE
? SHARING SPACE
MERGING
BOUNDARY ARITHMETIC COMPUTATION
SUMMARY AND TABLES

Decimal Notation

===
On the surface, boundary forms are spatial number-names, taking up an area rather than standing on a line. This means that positional notation is no longer available, the sequentially imposed by a string of digits becomes lost in the second dimension introduced by the spatial syntax. Instead, the dimension of space applied to a magnitude is recorded in depth of nesting. Crossing a boundary incurs a change of dimension. The functional zero is also no longer necessary, since the contents of a boundary can be empty, that is, not represented by a name at a place. Some examples in decimal notation:

0	
1	1
2	2
9	9
10	(1)
11	(1)1
12	(1)2
20	(2)
30	(3)
100	((1))

```
 253        ((2)5)3
 804        ((8))4
1000       (((1)))
4004       (((4)))4
```

===

===

The pure algebraic mathematics of boundaries [4] is based on the concept of *distinction*, or difference. It is constructed *de novo*, without reference to logical, set theoretic, relational, numeric, or categoric objects. Boundaries are strictly structural, representing only the abstract concept of difference, without requiring identification of the type of object being differentiated. Thus, boundary mathematics differs substantively from the conventional mathematics of strings.

2 Boundary Algebra

Composition of closed, non-intersecting planar curves, called *boundaries*, constructs a formal diagrammatic language, independent of an interpretation as logic. The alphabet is a singleton set of symbols consisting of the *empty boundary*, { ○ }, which is called a *mark*. A word consists of replicates of marks composed in a non-conventional manner. Since boundaries have both an inside and an outside, replicate symbols can be juxtaposed in two ways: on the inside of the original boundary and on the outside of the original boundary. Rather than one "concatenation" operator, there are two: SHARING is composition on the outside, while BOUNDING is composition on the inside. The formal language consists of the set of composable boundary forms.

2.2 Variary Operators

The functional basis set of the language of boundaries consists of two constructive operators, BOUNDING and SHARING. These diagrammatic operators are quite unconventional. An enclosing boundary can *bound* any number of forms, including none. Any number of forms can *share* a space, including none. In the example below, dots represent other single bounded forms. The explicit boundary encloses several forms, and while several others share the same external space.

The operators BOUNDING and SHARING do not have a specific arity, both are *variary*. Absence of a specific arity strongly differentiates the boundary formalism from modern algebra. The closest conventional description for a variary operator is that of a "single argument set function". Conventionally, relations are defined by a set of ordered pairs. Relational properties such as commutativity and transitivity describe the symmetries that the relation imposes upon its ordered arguments. A boundary, however, can contain any number of forms. The collection of forms contained within a boundary is *a priori* neither countable nor orderable. Since forms are not taken two-at-a-time, there is no concept of associativity. By construction, boundary mathematics does not support the numerical concept of arity nor the proximal concepts of commutativity and associativity. Boundaries are neither functions nor relations.

The structure of a boundary form is defined explicitly by the boundaries themselves. The space upon which boundaries are imposed is void; *void space* is neither metric, nor geometric, nor topological. Thus, there is no concept of geometric or topological localization that applies to boundaries. Boundaries can reside anywhere so long as they do not intersect other boundaries. Another consequence is that the boundary language includes no "empty word" other than the entire space that supports all boundaries. Since there is no specifically defined relative position for forms SHARING a space, an empty placeholder is not necessary. The idea of a null boundary is wrapped up within the ground symbol; absence of form is simply the inside of the mark, ().

Seen as a relation, a boundary distinguishes what is inside from what is outside. The most natural conventional interpretation of boundary forms is as a partial ordering, with BOUNDING providing a strict ordering, and SHARING providing an equivalence class of forms in the same space. Even so, since both BOUNDING and SHARING are variary, a relational interpretation is difficult. The conceptualization represented by the boundary language does not support conventional relational properties such as reflexivity, symmetry and transitivity. Conventionally, variary relations are universal.

2.3 Pattern-Templates and P
attern-Equations

Let the set of capital letters, {A,B,...}, provide variables that can stand in place of any boundary form, including many forms and no forms, and the set of small letters, {a,b,...}, provide variables that can stand in place of either a mark, or the absence of a mark. When variables are included within boundary forms, for example (A ((B))), forms can be used as *pattern-templates* to identify specific structure within other forms. The pattern-template (A ((B))) matches ((()) (()())), with A=(()()) and B=*void*. Pattern matches are spatial rather than textual, while pattern variables can match many forms within a space, as well as matching no forms at all.[2]

Similar to the idea of extensionality in set theory, two bounded forms are identical only if they enclose identical contents. A *pattern-equation* is an assertion that two structurally different patterns are equal; forms that match either pattern have the same value (modulo the asserted equation), partitioning the set of forms into equivalence classes. The *semantics* of the boundary language is defined by pattern-equations that assert specific patterns, or pattern-templates, to be equivalent. A set of pattern-equations create a particular boundary algebra.

Consider the following two pattern-equations from Laws of Form as providing an evaluation function for forms not containing variables. The two equations reduce any form either to a mark, (), or to the absence of a mark, thus establishing two equivalence classes, mark-equivalence and void-equivalence.[3]

```
( ) ( ) = ( )          SHARING EVALUATION
 (( ))  =              BOUNDING EVALUATION
```

===

BOUNDARY ARITHMETIC AND DEPTH-VALUE NOTATION

Unit ensembles provide the advantages of clarity of structure and ease of modeling. Of particular importance, they are an example of spatial mathematics that demonstrates the advantages of a spatial formalism without concepts of associativity, commutativity, and arity. Equally as important, unit ensembles closely represent the elementary number processes used by young children. The cost of this clarity is ease of reading, since an ensemble must be counted in order to determine its value. As well, the representation of large ensembles is unwieldly. We provide an efficient notation quite similar to place-value notation while maintaining the models of addition as fusion and multiplication as substitution. The efficiency covers representing, reading and computing with spatial numbers.

PLACE-VALUE NOTATION AS SYNTACTIC SUGAR

The structures exhibited during the axiomatization of unit ensembles are not change when the abbreviation of numerals are attached. That is, whether or not we use unit ensembles or conventional base-10 place-value notation, the mathematics of integers does not change. Place-value notation assumes nothing from group theoretic structures since it is simply a naming convention, also called syntactic sugar, to make numbers more easily readable. Conversely, nothing in group theory requires place-value notation.

We are left with the remarkable observation that the axioms of unit ensemble arithmetic are equally valid for place-value arithmetic, only specific implementation details change. What then is the added value contributed by a group theoretic formalization of arithmetic and elementary algebra? The answer is easy, since modern algebra lays the groundwork for the magnificent edifice that is advanced mathematics. We refine the question: what is the added value contributed by a group theoretic formalization of arithmetic and elementary algebra to mathematical novices such as elementary and high school students? Three directions need to be considered:

1) modern algebra prepares novices for an understanding of higher mathematics,
2) modern algebra is the only viable way to formalize mathematics, and
3) the organizational principles of modern algebra provide a better understanding of mathematical thinking in general.

We have limited our question to novices will presumably never need the sophistication of modern algebra. Would anything be lost if group theory were taught first in college? What would be gained may be an improvement nationally in mathematics performance and understanding, if the alternative of doing without modern algebra were acceptable. This brings us to the second direction,

that there is no alternative. We have labored in the preceding pages to show that there is indeed an alternative, and that alternative is more concrete, more intuitive, and computationally more efficient.

We will also take a stronger position that the current alternative of teaching concepts such as commutativity, associativity, and arity are dysfunctional, a source of math problems rather than a solution. We propose that mathematics understanding and performance can be improved by returning to the axiomatics that is understood in first grade, particularly concrete manipulatives with parallel, variary operations of fusion and substitution. Fusion embodies the Additive Principle, while substitution is the simple skill of forming groups in one-to-one correspondence to units. We propose that the concrete/abstraction comprehension gap encountered by students as they transition from direct manipulation of quantity to abstract symbolic manipulation of arbitrary tokens with potentially dysfunctional theoretical organization can be addressed by eliminating the transition to the abstraction of modern algebra.

Finally we will argue that mathematical thinking is enhanced by making mathematics more direct, more visceral, more spatial. The principles of mathematical structure that support manipulation of strings composed of arbitrary tokens are not the principles that lead to mathematics understanding. They do indeed lead to efficiency in digital computers, so our position can be summarized by saying that minds are not computers, and bodies are not symbolic. Understanding can be achieved by uniting mathematical law with physical experience.

The consequences of such a position include

-- use calculators for all token manipulation tasks. Associate understanding with physical manipulation.

-- embrace formality but broaden the forms of representation

Unit Ensembles Expressed in Place-value Notation

We construct a union of conventional, and universal, place-value notation and unit ensemble arithmetic, called place-value ensembles.

Name unit ensembles with plae-value numeral names

Place-value Fusion

Accept the rules of digit addition as name conversions. Eg

2+4 = 6
8+9 = 17

We need to provide a structure so that fusion of place-value numerals will represent sums.

17 == (1)7

12 == (1)2

17 + 12 == (1)7 | (1)2

Use boundary notation

(1)7|(1)2 = (1)7 (1)2 = (1)(1) 7 2 = (1 1) 7 2 = (2)9

Place-value Substitution

Accept the rules of digit multiplication as name conversions.

2*4 = 8
8*9 = 72

17*12 = 204

17*12 == [(1)7 ∂ (1)2]

∂ stands for a generalized unit that includes the multiplication table. Curly braces are highlights.

[(1)7 ∂ {1}2] = {(1)7}(2)(1)4 = {(1)7 2 1}4 = {(1)(1)}4 = {(1 1)}4 = {(2)}4

Base-10 Notation

Conventional numerical notation incorporates three principles of simplification:

- specific tokens represent specific ensembles
- place-value notation permits reuse of tokens to represent larger ensembles
- binary arity limits operator rules to combining two numbers at a time

In the case of base-10 notation, we have ten numeral digits, 0-9, each of which is an abbreviation for a small ensemble of units. We then place replicate digits in a string, assigning each position to the left of the unit a power of the base 10. Place-value notation uses increasing multiples of the base 10 to express larger groups:

3258: (3x1000) + (2x100) + (5x10) + (8x1)

The cost of using specific digits is that they require number facts (such as 4 + 5 = 9) that must be memorized in order to achieve computation. The binary base-2 system requires 4 (2x2) facts and can thus take advantage of place-value notation with little overhead of memorization. For this reason, it is convenient for computer hardware implementations of numerical computation. The base-10 system requires 100 (10x10) number facts. The burden of memorization is significantly lightened by symmetries such as zero rules and commutativity.

The number of number facts is limited by binary operators that combine exactly two numbers per operation. Should we need to join three numbers, we apply the operator twice. This limits number facts to covering only binary combinations; there are n^2 facts for a base-n notation. We could use a ternary base-10 addition, for example, that would require only one operation, but then we would need to recall 1000 (10x10x10) facts. Costs associated with binary arity include multiple operations and a need for associative and precedence rules.

In place-value notation, digits are maintained in a strict sequential position; calculation then includes techniques for interfacing adjacent places, these are known as called "carrying" and "borrowing". The cost of a positional notation system includes keeping track of positions (i.e. aligning numbers during transformation) and transforming digital overflows into an adjacent position.

Hybrid Unit Ensembles

Any of the three techniques that make conventional arithmetic computation simpler can be combined with the unit ensemble system. It makes little sense to limit unit ensembles to binary operations, since variary operations are at the heart of spatial forms. Likewise, adoption of positional notation, especially

place-value notation, injects total ordering into the spatial system that explicitly avoid total orderings. We can, however, introduce numerals to abbreviate ensembles. This idea is ancient and is best exemplified today by the antiquated system of Roman Numerals.

Both Roman Numeral abbreviations and base-10 decimal digits are easily accommodated by the methods of Spatial Arithmetic, since these are simply nicknames for ensembles of particular sizes. The cost is having to include tha simple number facts for adding and multiplying digits between 0 and 9. The fundamental transformation principles of addition by sharing a space, and multiplication by substitution still apply. For example

```
2  ==  ●●
3  ==  ●●●
5  ==  ●●●●●
6  ==  ●●●●●●
2+3 = 5   ==   ●●|●●● = ●●●●●
2*3 = 6   ==   [●● ● ●●●] = ●●●●●●
```

Here, rather than using fusion or substitution, we retrieve the number fact associated with the operation, and then re-associate the result with its unit ensemble representation. Since we have these numerals available, we may as well write:

```
2+3 = 5   ==   2|3 = 5
2*3 = 6   ==   [2 1 3] = 6
```

We immediate difficulty is that we do not have an easy way to handle multi-digit numbers:

```
5|6 = ●●●●●●●●●●
```

An example of a base-10 boundary-number:

```
3258:          ((((3) 2) 5) 8
```

===

depth-value shunts compute to standardization

BOUNDARY-NUMBERS

The boundary-numbers of Spatial Arithmetic are spatial pictures rather than strings of digits. These techniques were first published in 1995 by Louis H.

Kauffman, a Full Professor of Mathematics at the University of Illinois at Chicago. Spatial Arithmetic is dynamic, it shows the structure and transformation of numbers directly without the bookkeeping associated with place values. As an analogy, if the techniques of Spatial Arithmetic applied to words, we would be able to read a word regardless of the ordering of the letters!

Two or more boundary-numbers are added together by placing them in the same space, just like stroke arithmetic and Roman numerals. Unlike stroke arithmetic, depth-value notation provides the same advantages in readability as does place-value notation.

Instead of addition operations, boundary-numbers sharing a space are simplified by a standardization process that results in the minimal spatial form that is easiest to read. This standardization process is extremely simple, consisting of two rules, "Join Units" and "Join Boundaries". For clearer presentation, the two rules are presented below in a binary rather than decimal base, and using a textual notation that permits recording boundary-numbers on typographical lines.

1 = • 2 = (•) 3 = (•)• 4 = ((•)) 5 = ((•)) 6 = ((•)•) ...

1 + 1 = 2: • • = (•) JOIN UNITS

---> add
sub(borrow) <---

2 + 2 = 2(1+1): (•)(•) = (• •) JOIN BOUNDARIES

---> mult
divide <---

Boundary-numbers are multiplied simply by substituting the number being multiplied for each unit (i.e. each •) in the base number.

2 x 3 = 6: (•) x (•)• ==> ((•)•)

Substitution alone achieves multiplication. The same standardization process, using the same two simple rules above, converts the result of multiplication into an easier to read form. Subtraction and division are also achieved using the same two rules, applying them in reverse, from right to left.

===

Canonical Form of Numbers

Every number has a unique canonical form, relative to another number called a base.

The base is a number with a special representation. It is used to collect other numbers into groups. The special representation is a spatial depth, (...).

When () is empty, the base is the unit:

• = ()

The canonical form relative to the unit is the collection of unit dots.

When (...) is not empty, it represents the base b, multiplying the contents c.

(•) = "subst •• for (•) in (•)" = •• (base ••)

Examples:

(•) = "subst ••••••••••• for • in •" = ••••••••••• (base ••••••••••• 10)

(•)• = "subst •• for (•) in (•)•" = ••• (base ••)

The canonical form relative to base b is a composite of base and dot forms.

Definition of Base ()

Equality

Two numbers are equal if they have the same representation wrt the base

READING BOUNDARY INTEGERS

The absence of form is 0, while the mark is 1. Each level of nesting doubles the value of the nested contents. The value of a boundary form is the sum of the forms sharing a space. Merging provides a smooth transfer between unit notation (each unit is represented by a single mark) and depth notation. Here, four is provided as an example of merging:

• • • • = (•) (•) = (• •) = ((•))

1+1+1+1 = 2*1 + 2*1 = 2*(1+1) = 2*2*1

The conventional interpretation is vertically aligned, units corresponding to 1; boundaries corresponding to 2.

Here is a reading of 14:

(((•) •) •)

2*(2*(2*1 + 1) + 1)

===
SUBTRACTION

Rings are transparent to polarity, such that

-(A) = (-A)

Thus, all that needs to be introduced is the polar unit, ◊, which collapses a common unit, •, into the void:

• ◊ =

When a given boundary integer, ((•) •) for example, possesses the polar property, all units in that number are changed to ◊, the polar unit, generating ((◊) ◊) in the example.

A polar number can also be interpreted as multiplication by -1, converting a unary property into a binary relation with a constant. Multiplication is defined as a substitution operation: Substitute ◊ for • in A (which can read as "Substitute -1 for 1 in A"

The addition rule for unit • and ◊ is to delete both. For all other cases, the rules for addition are identical, blind to the type of unit. For boundary integers, the distinction between a polar number and a number multiplied by -1 is one solely of time, of when the form is examined.

A form composed of bounded ◊ units has the ◊ property; the ◊ property could be lifted to a universal of the entire form:

"Substitute ◊ for • in A" is "Read A with ◊ units".

Conversely, a universal property can be mapped over a form to apply only to units. Forms sharing a space consequently do not compose differently for different unit polarities.

A special case arises when A already contains ◊ units. This could occur when the polarity of a ◊-form is reversed, when a form is in process of additive

computation and not yet homogenous in units, or when nested forms have a diversity of polarities (again prior to simplification).

It is sometimes convenient to ascribe a properties to rings, such as their base unit and polarity. Here we use subscripts that proceed the ring they describe:

$$\Diamond(a) = (\Diamond a)$$

Recalling that the rings in a number are blind to unit polarity, only four unit substitution varieties can occur:

Substitute ● for ● in A	common *-substitution
Substitute ◊ for ● in A	common *-substitution
Substitute ● for ◊ in A	inverting ◊-substitution
Substitute ◊ for ◊ in A	inverting ◊-substitution

◊-substitution is unusual in that it changes the polarity of the unit being substituted. Thus, when ● is substituted for ◊, the resulting unit is not-●, i.e. it is ◊. Similarly, when ◊ is substituted for ◊, the resulting unit is not-◊, i.e. it is *.

The difference between the polarity property of an expression, and the act of subtraction is:

$\Diamond$A is "Substitute ◊ for both ● and ◊ in A"

A - B is A + "Substitute ◊ for both ● and ◊ in B"

The intent is to separate polarity from numerical simplification. It is convenient to fully encode polarities at the beginning of a simplification, resolving them when they are read and substituted into the initial representation of the problem. This perspective treats the conventional difference between an inverse operation and polar property as syntactic, as an artifact of conventional notation rather than concise mathematical formulation.

===
Thus, for example, given this atomic transcription:

	5	((●)) ●	
	7	((●) ●) ●	
	-	property ◊:	
		change polarity of all units, delete property	
-(5-7)			
	-7	((◊) ◊) ◊	property ◊
	(5-7)	((●)) ● ((◊) ◊) ◊	operation (sharing space)

```
                -(5-7)      ((◊)) ◊  ((●) ●) ●      global property ◊
                  =         (●)                     calculation
```

Any binary operation of subtraction can be incorporated into the boundary encodement as a property of the second argument. An alternative reading:

```
      -(5-7)
                 (5-7)      ((●)) ●  ((◊) ◊) ◊      operation (subtract 7)
                -(5-7)      ((◊)) ◊  ((●) ●) ●      global property ◊
                  =         (●)                     calculation
```

The distinction is that the polarity of the 7 was compiled with the form of 7 rather than through subtraction, the inverse operation for addition. The double inversion below illustrates an unnecessary specification:

```
      -(5 - -7)
                 -7         ((◊) ◊) ◊               property ◊
                --7         ((●) ●) ●               property ◊
                 (5--7)     ((●)) ●  ((●) ●) ●      share space,
                                                      delete double inversion
                -(5--7)     ((◊)) ◊  ((◊) ◊) ◊      global property ◊
```

===

The advantage of compiling additive inversions is that the addition/subtraction procedure can then be blind to polarity. Due to the annilihation rule,

```
                ● ◊ =
```

only one type of unit will survive in the canonical boundary form of the number. Communication of polarity through levels is provided through:

```
                (a) ◊ = a
```

Since a ring is simply a unit with contents, ◊ annihilates rings as well as units. The annihilation process can occur at each level of nesting concurrently. No priority need be given to unit or ring annilihation, since both action reduce the number of ◊ by one.

The equifinal result of simplification is a minimal boundary structure. From the perspective of a ◊-space, ● is an identical annihilater. The initial encodement does not produce a single type form, rather it partitions the mixture of units types into two phases. The first removes syntactic structure, the second removes mixed polarity due to the action of addition on different types.

When an expression is negative, as in -(7+5), the negative property remains as "Substitute ◊ for both ● and ◊ in A."

===

Here is an example of varieties of four units of different types composed in the same space:

```
● ● ● ●  =  (●)(●)  =  (● ●)  =  ((●))
● ● ● ◊  =  * *             =   (●)
● ◊ ● ◊  =
         =  (●)(◊)  =  (● ◊)  =   ( )   =
● ◊ ◊ ◊  =     ◊ ◊          =   (◊)
◊ ◊ ◊ ◊  =  (◊)(◊)  =  (◊ ◊)  =  ((◊))
```

===

===

Composition of closed, non-intersecting planar curves, called boundaries, constructs a formal diagrammatic language, independent of an interpretation as logic. The alphabet is a singleton set of symbols consisting of the empty boundary, { ○ }, which is called a mark. A word consists of replicates of marks composed in a non-conventional manner. Since boundaries have both an inside and an outside, replicate symbols can be juxtaposed in two ways: on the inside of the original boundary and on the outside of the original boundary. Rather than one "concatenation" operator, there are two: SHARING is composition on the outside, while BOUNDING is composition on the inside. The formal language consists of the set of composable boundary forms.

The functional basis set of the language of boundaries consists of two constructive operators, BOUNDING and SHARING. These diagrammatic operators are quite unconventional. An enclosing boundary can bound any number of forms, including none. Any number of forms can share a space, including none. In the example below, dots represent other single bounded forms. The explicit boundary encloses several forms, and while several others share the same external space.

INSIDE AND OUTSIDE

The fundamental innovation is to introduce a representation that has both an inside and an outside. A circle () is the prototypical planar boundary form. String tokens have only an outside, so that concatenation of string tokens is defined as adjacency on the outside. Boundary symbols, such as circles, can be composed on the outside, () (), and on the inside, (()).

One consequence of "two sided" symbols is that they can be interpreted both as objects (when composed on the outside), and as operators (when composed on the inside). It is important to accept that a boundary symbol is atomic, and not a relation between inside and outside. As an atomic structure, a boundary symbol

is concurrently object and operator, the specific interpretation is lifted from the representation itself, becoming a semantic choice. That is, in boundary mathematics, there is no explicit differentiation between object (member of a domain or codomain) and operator (function assigning correspondence between domain and codomain members).

When a boundary form is read as an object, transformation can be defined by substitution of equivalent objects. Objects can be generalized to patterns by including variables to represent arbitrary forms. For example, the idempotency pattern gives permission to delete all replicate forms sharing a space. Below, this pattern is illustrated for atomic objects and for patterns:

() () = ()

A A = A

SHARING SPACE

String tokens are situated, left and right adjacency are clearly defined. Boundary tokens, in contract, rest in space. A planar boundary token

In Laws of Form, Spencer-Brown presents boundary logic, an interpretation of boundary mathematics for logic. The interaction of boundaries maps to elementary logic via the two axioms in LoF:

CALLING ()() = ()

CROSSING (()) =

Boundary mathematics is a generic technique, the interpretation changes when the initial axioms are different. Lou Kauffman first introduced me to the axioms of boundary integers:

UNIT MERGE ()() = (())

BOUNDARY MERGE (...)(...) = (... ...)

The ellipses stand in place of any existing content; thus UNIT MERGE is a rule for empty containers, while BOUNDARY MERGE is a rule for non-empty containers.

Boundary math is a minimal foundational formalism, so it illustrates essential differences between systems, here logical and integer arithmetic are contrasted.

MERGING

For boundary integers the empty container, or mark, is not idempotent (i.e. two sharing a space do not equal one), CALLING does not apply. Instead, two marks sharing a space become nested. This generates a depth notation (as opposed to conventional place notation for numbers), the contained form is doubled by each container.

Another fundamental difference between logic and numerics is that numerical forms do not degenerate into void, CROSSING does not apply. For boundary integers non-empty containers do follow the rule of CALLING, merging non-void rather than void contents.

Because marks are atomic elements, in boundary integers it is more convenient to make them solid: ( ) = •

UNIT MERGE • • = (•)

BOUNDARY MERGE (•)(•) = (• •)

===

The pattern of the divisor A can be converted into a unit at any depth of nesting of R, in any order. Thus, pattern-matching can occur in parallel across levels. There will be at most one matching pattern in R whenever R is in canonical form.

Outer rings can be pruned by cancelation from both A and R, for example:

```
          ((•)•)   /   (•)
-->        (•)•    /    •
```

To identify patterns, it is necessary to decompose R using the reverse of rules for simplification:

```
 (•)    -->   • •
(a b)   -->  (a)(b)
```

BOUNDARY ARITHMETIC COMPUTATION

The additive principle is that the result of a sum can be determined by counting up the resultant set of tokens.

To ADD boundary integers, place them in the same space. In contrast to conventional place notation, any number of forms can be added concurrently, while the result of placing boundary integers in the same space can immediately be read as a sum.

MERGING generates a canonical form with the fewest number of units and boundaries. For example:

```
    7       +       5          =          12

((•) •) •        • ((•))       =   ((•) •) • • ((•))
```

SUMMARY

===
DIFFERENT BASES

The UNIT MERGE presented above is binary, base two. Any base can be used, here is base six:

```
      • • • • • • = (•)
```

Collections of one through five units do not merge, they are independent entities, usually represented by an abstract token:

```
     2:      • •
     3:    • • •
     4:  • • • •
     ...
===
```

11. Kempe, A.B. Memoir on the Theory of Mathematical Form. Philosophical transactions of the Royal society of London 177 (1886) 1-70.

1. Barwise, J., Etchemendy, J. Heterogeneous Logic. In: Allwein, G, Barwise, J. (eds.) Logical Reasoning with Diagrams. Oxford University Press (1996).
2. Shin, S. The Logical Status of Diagrams. Cambridge University Press (1994).
3. Shin, S. The Iconic Logic of Peirce's Graphs. MIT Press (2002).
4. Hammer, E.M. Logic and Visual Information. CSLI, Stanford (1995).
5. Jamnik, M. Mathematical Reasoning with Diagrams. CSLI, Stanford (2001).

===

FROM DoE 0711

II.3 Boundary Integer Arithmetic

The fundamental innovation in boundary integer arithmetic is to introduce a representation that has both an inside and an outside. A circle, () -- here represented typographically by a pair of parentheses -- is the prototypical planar boundary form. String tokens have only an outside, so that concatenation of string tokens is defined as adjacency on the outside. Boundary symbols, such as circles, can be composed on the outside, () (), and on the inside, (()). A representation with an inside permits a different type of "place" to record the power of a base. Boundary forms use depth-value notation. Crossing a boundary changes the degree of the base, just as moving one digit right or left changes of the degree of the base in place-value notation.

Boundary forms are spatial number-pictures, taking up an area rather than standing on a line. Since boundary forms occur in space rather than in a line, the concept of a linear place-notation is simply not part of the structure of these numbers.

Boundary forms are necessarily diagrammatic, boundary arithmetic is not isomorphic to place-value arithmetic. For example, a representation of zero is no longer necessary, since the contents of a boundary can be empty; that is, an empty boundary contains nothing, the zero concept is structural rather than symbolic.

There are two structural transformation rules for boundary arithmetic, here expressed in a degenrate typographical notation for convenience:

UNIT JOIN (base 2)	• • = (•)	1+1 = 2x1
BOUNDARY JOIN	(•)(•) = (• •)	2x1 + 2x1 = 2(1+1)

These rules convert a unit arithmetic forms (collections of identical members) into a form expressed in depth-value notation, just as "factoring out powers of the base" converts unit forms into a place-value notation. The value of a boundary form is the sum of the forms sharing a space (the additive principle). Unit Join above defines each level of nesting as doubling the value of the nested contents. Boundary Join is the distributive rule in its most elementary form. It is independent of contents and thus independent of base.

Note that Unit Join above is binary, analogous rules for standardization of base-10 boundary forms would be:

UNIT JOIN (base 10)	•••••••••• = (•)	1+1+1+1+1+1+1+1+1+1 = 10x1

BOUNDARY JOIN (a)(b) = (a b) 10a + 10b = 10(a+b)

Naturally, the names of digits can be included in base-10 boundary arithmetic. Number facts (i.e. the addition table) would be required to convert the symbolic names of digits into sums, while depth-value notation would handle the management of carries and borrows. For example, the base-10 place-value numeral 734 would be represented as ((7)3)4 in base-10 depth-value notation augmented with digit symbols.

For clarity and convenience, we will continue to present the operations of boundary arithmetic in base 2. Below, unit-four is converted into depth-value four by application of the Joining rules. A conventional interpretation is vertically aligned: units correspond to +1, boundaries correspond to 2x.

• • • • = (•) (•) = (• •) = ((•))

1+1+1+1 = 2*1 + 2*1 = 2*(1+1) = 2*2*1

Here is a reading of boundary 14:

(((•) •) •)

2*(2*(2*1 + 1) + 1)

II.3.1 Addition Is Sharing Space

To ADD boundary integers, place them in the same space, similar to addition in unit arithmetic. In contrast to conventional place notation, any number of forms can be added concurrently, while the result of placing boundary integers in the same space can immediately be read as a sum.

Joining generates a canonical form with the fewest number of units and boundaries. Thus, the operation of boundary addition maintains the simplicity of the Additive Principle, while the two joining operations convert sums into a minimal form with the advantages of a consistent base notation. For example:

7 + 5 = 12

((•) •) • • ((•)) = ((•) •) • • ((•))

111 + 101 = 1100

The above sum, consisting of four separately bounded forms, can be read as

6+1+1+4. Joining reduces the number of units and marks from 9 to 5. Figure 3 shows this simplification. This sum is not analogous to addition of place-value notations in base-2, as is shown in Figure 4.

===

```
((•) •) • • ((•))
((•) •) (•) ((•))
((•) •   •   (•))
((•)  (•)    (•))
((•    •      •))
(( (•)        •))
```

===

II.3.2 Multiplication Is Unit Substitution

To MULTIPLY boundary integers, substitute one form for every unit in the other form. This is the same process as multiplication in unit arithmetic, however depth-value notation manages representational complexity. In contrast to conventional place-value notation, any number of forms can be substituted concurrently, while the result of substituting boundary integers for units can immediately be read as a product. Figure 5 shows two examples. The case of 3x2 requires two substitutions and an additional simplification step relative to 2x3, since the representation of 3 includes two units, while the representation of 2 incorporates only 1 unit. Figure 6 shows 5x7 in boundary notation. In Figure 6, the substitution immediately yields three separate bounded forms, which can be read as 28+6+1. Joining these standardizes the boundary form of 35 to have the minimal number of components (units and boundaries). Figure 7 shows 7x5. The substitution immediately yields three separately bounded forms, can be read as 30+4+1. Joining reduces the product to the same form as 5x7.

===

1 x 3

```
  •       (•) •       Substitute (•)• for • in •
                        •>(•)•I•   S[(•)•,•,•]
 (•) •
```

2 x 3

```
 (  •  )    (•) •       Substitute (•)• for • in (•)
                        •>(•)•I(•)   S[(•)•,•,(•)]
 ((•) •)
```

3 x 1

```
 (•) •       •          Substitute • for • in (•)•
                        •>•I(•)•   S[•,•,(•)•]
 (•) •
```

```
3 x 2
              ( • )  •     •         Substitute (•) for • in (•)•
                                        •>(•)I(•)•   S[(•),•,(•)•]
              ((•)) (•)  -->  ((•) •)
```

The case of 3x2 requires two substitutions and an additional simplification step relative to 2x3 since 3 incorporates two units while 2 incorporates only 1. Thus although multiplication is "commutative" in final result, it is non-commutative effort. Addition (i.e. sharing space) commutes in both effort and result.

Additional examples:

```
           •>((•)•)•I((•))•   S[((•)•)•,•,((•))•]

          5       x       7          =          35

        ((•)) •       ((•) •) •
```

Substituting (the two replicas of boundary 7 are highlighted with overbars):

```
                                    _________     _________
                                (( ((•) •) • ))  ((•) •) •
```

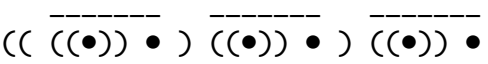
The above sum, consisting of three separately bounded forms, can be read as 28+6+1. Merging reduces the number of units and marks from 12 to 8:

```
           ((((•) •) •)) ((•) •) •
           ((((•) •) •)   (•) •) •
           ((((•) •) •     •) •) •
           ((((•) •)   (•)  ) •) •
           ((((•) •     •)  ) •) •
           ((((•)   (•)  )  ) •) •
           ((((•     •)  )  ) •) •
           ((((  (•)  )  )  ) •) •
```

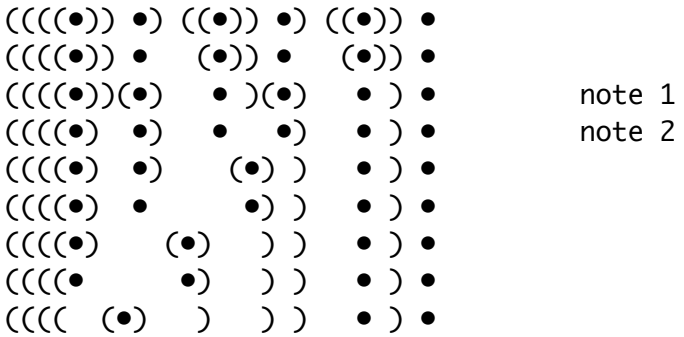
Alternatively:

```
           •>((•))•I((•)•)•   S[((•))•,•,((•)•)•]

          7       *      5          =          35

        ((•) •) •      ((•)) •
```

Substituting (the three replicas of boundary 5 are highlighted with overbars):



The above sum, consisting of three separately bounded forms, can be read as 30+4+1. Merging reduces the number of units and marks from 12 to 8:



Note 1: This linear rearrangement is an artifact of typing spatial boundary forms on a line, just as the apparent fragmentation of a contiguous boundary into right and left parentheses is an artifact of typography.

Note 2: At this point 5x7 and 7x5 become identical. The comparative effort:

	5x7	7x5
substitutions	2	3
multiplication effort	2(5-1) = 8	3(4-1) = 9
ring merges	4	5
unit merges	3	3
simplification effort	7	8

===

The mechanism of boundary multiplication is function composition rather than the additive composition of place-value notation. This makes boundary multiplication mathematically more abstract while at the same time broadening the constructive definition of multiplication (such as Peano's axioms, which rely on recursive addition to achieve multiplication).

II.3.3 Subtraction Is Making Nothing

Negative numbers attach a polarity to the whole numbers, so that different poles annihilate:

A + -A = 0

"Negative" numbers are a natural consequence of the Rule of Equational Identity:

A = A
A - A = 0

In boundary systems, zero is non-represented, it is the absence of a mark. A form combined with its identical but opposite form of opposite is equal to nothing at all.

A -A =

The negative unit, represented here as ◊, collapses a positive unit, •, into the void:

• ◊ = CANCELLATION

When a given boundary integer, ((•) •) for example, possesses the negative property, all units in that number are changed to the negative unit ◊, generating ((◊) ◊) in the example.

A negative number can also be interpreted as multiplication by -1, converting a unary property into a binary relation with a constant. Since boundary multiplication is a substitution operation, subtraction can be subsumed by multiplication:

Substitute ◊ for • in A which can read as "Substitute -1 for 1 in A"

Other than the Cancelation Rule, the operation of addition and the joining rules of simplification are blind to the type of unit.

II.3.4 Division Is Finding Patterns

Multiplication calls for substitution of a form A for each unit in a second form B. Which form is substituted into which (A into B or B into A) is irrelevant to the result, but relevant to the effort (number of steps) during simplification.

Division calls for substitution of a unit, •, for each form A within a second form P. The pattern of the divisor A can be converted into a unit at any depth of nesting of P, in any order. Thus, boundary pattern-matching can occur in parallel across levels.

To identify patterns, it is necessary to decompose P using the reverse of the two Join rules:

```
            UNJOIN UNITS             (•)   -->   • •
            UNJOIN BOUNDARY         (a b)  -->  (a)(b)
```

Selection of decompositions is the only action that makes division (and factorization) more difficult than multiplication.

```
===
egs  at end
===
```

```
=========================================================================
```

UNIT BEHAVIOR

```
      • ◊         =     ><
      • •         =     (•)
```

RING BEHAVIOR FOR DIFFERENT CONTENTS

```
      ( )         =     ><
      (a b)       =     (a) (b)
```

RING OPERATIONS FROM THE CONTEXT

only the presence of ◊ will trigger (a b) --> (a)(b)

```
      (  •) ◊     -->   ( ) •   -->   •
      (a •) ◊     -->   (a) •
```

```
=========================================================================
```

II.4 Varieties of Decomposition Strategies

Figure 8 illustrates various decomposition strategies. These strategies are at the base of algorithmic complexity for operations and for reading. Unit form (a number is represented by the sum of single units) is the ultimate additive decomposition, while prime factorization (a number is represented by the product of primes) is the ultimate multiplicative decomposition.

Unit decomposition results in unit arithmetic, which is very easy to transform and very hard to read. Base-10 polynomial decomposition results in place-value notation, which is easy to read and relatively easy to transform. Base-2 polynomial decomposition results in the same place-value notation with a differing base. Prime factor decomposition makes multiplication easy, addition difficult, and reading very difficult, since it abandons the uniformity of the base. Boundary notation makes reading slightly more difficult than place-value notation, while turning addition and multiplication into simple operations.

Figure 3: Adding and then Simplifying Boundary 7+5

```
((•) •) •   +   • ((•))    =     ((•) •) • • ((•))
```

```
((•) •) • • ((•))          2(2+1) + 1 + 1 + 2(2)
((•) •) (•) ((•))          2(2+1) +   2   + 2(2)
((•) •   •  (•))           2(2+1  +   1   +   2)
((•)  (•)   (•))           2(2+   2       +   2)
((•    •     •))           2(2(1   + 1    +   1)
(( (•)       •))           2(2(2          +   1)
```

Figure 4: Simplifying 7+5 in Place-value notation (base 2)

```
1x2^2 + 1x2^1 + 1x2^0 + 1x2^0 + 1x2^2
1x2^2 + 1x2^1 +     1x2^1     + 1x2^2
1x2^2 +     1x2^2             + 1x2^2
1x2^3                     + 1x2^2
```

Figure 5: Examples of Boundary Multiplication

2 x 3

```
( • )    x  (•) •          Substitute (•)• for • in (•)

((•) •)
```

3 x 2

```
( • ) •   x  (•)           Substitute (•) for • in (•)•

((•)) (•)  -->  ((•) •)
```

Figure 6:
Multiplying 5 by 7 in Boundary Notation, and then Simplifying

```
    5       x       7          =          35


                                          _________     _________
  ((•)) •   x   ((•) •) •      =    (( ((•) •) • ))  ((•) •) •
```

(The two replicas of boundary 7 are highlighted with overbars)

Joining:

```
           ((((•) •) •)) ((•) •) •
           ((((•) •) •)   (•) •) •
           ((((•) •) •     •) •) •
           ((((•) •)   (•)  ) •) •
           ((((•) •     •)  ) •) •
           ((((•)   (•)  )  ) •) •
           ((((•     •)  )  ) •) •
           ((((  (•)  )  )  ) •) •
```

Figure 7:
Multiplying 7 by 5 in Boundary Notation, and then Simplifying

```
    7       *       5          =          35


                                          _______   _______   _______
  ((•) •) •       ((•)) •      =    (( ((•)) • ) ((•)) • ) ((•)) •
```

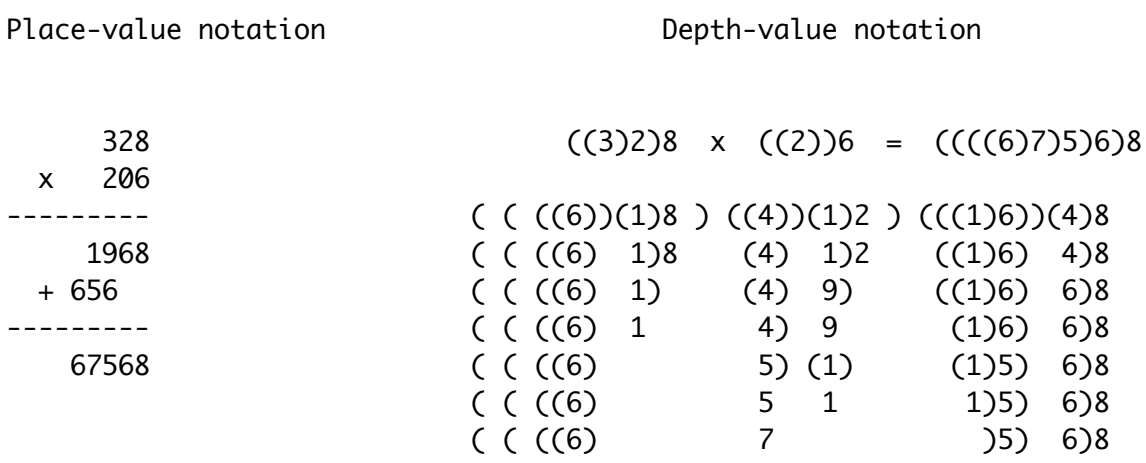
Joining:

```
           ((((•)) •) ((•)) •) ((•)) •
           ((((•)) •   (•)) •   (•)) •
           ((((•))(•)   • )(•)   • ) •        note 1
           ((((•)  •)   •   •)   • ) •        note 2
           ((((•)  •)    (•) )   • ) •
           ((((•)  •      •) )   • ) •
           ((((•)    (•)   ) )   • ) •
           ((((•      •)   ) )   • ) •
           ((((  (•)   )   ) )   • ) •
```

Note 1: This linear rearrangement is an artifact of typing spatial boundary forms on a line.

Note 2: At this point 5x7 and 7x5 become identical.

Figure 14: Multidigit decimal multiplication problem in place-value and depth-value notation

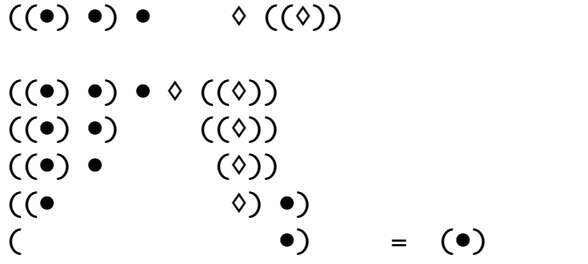


Other examples:

2-1

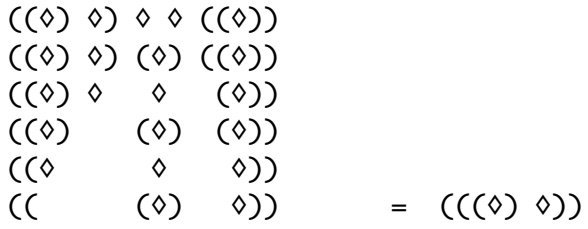
1-2

7-5

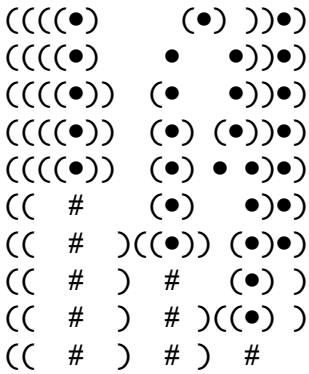


5-7

-7-5



```
2/2                                  (•)>•I(•)     S[•,(•),(•)]
           (•)       (•)           Substitute • for (•) in (•)
            #

6/2                                  (•)>•I((•)•)     S[•,(•),((•)•)]
           ((•)  •)    (•)         Substitute • for (•) in ((•)•)
           ( #   •)
           ( # )(•)
           ( # ) #

6/2 --> 3/1                            •>•I(•)•     S[•,•,(•)•]
           ((•)  •)    (•)
            (•)  •      •
           ( # ) #

35/7                    ((•)•)•>•I(((((•)))•)•   S[•,((•))•,(((((•)))•)•]
           (((( (•) ) ) )•)•
           ((((•   •) ) )•)•
           ((((•) (•) ) )•)•
           ((((•)•   •) )•)•
           ((((•)•) (•) )•)•
           ((((•)•)•   •)•)•
           ((  #       •)•)•
           ((  #  )   (•)•)•
           ((  #  )) ((•)•)•
           ((  #  ))   #

35/5                     ((•))•>•I(((((•)))•)•     S[•,((•)•)•,(((((•)))•)•]
           ((((  (•)  ) ) )•)•
           ((((•     •) ) )•)•
           ((((•)   (•) ) )•)•
           ((((•)  •   •) )•)•
           ((((•))(•   •) )•)•
           ((((•))(•) (•) )•)•
           ((((•))• • (•) )•)•
           ((  #    •  (•))•)•
           ((  #  )(•)((•))•)•
           ((  #  )(•)  #   )•
           ((  #  ) #   (•)  )•
           ((  #  ) # )((•)  )•
           ((  #  ) # )  #


35/5
           ((((   (•)   ) ))•)•
           ((((•       •) ))•)•
```



LARGE EXAMPLE

```
      204            ((((((((•)•) ) )•)•) )                7 rings 4 units
      358           (((((((((•) )•)•) ) )•)•)              8 rings 5 units
    73032   ((((((((((((((((((•))))•)•)•)))•))•)))•)))     16 rings 7 units

    358x204
      (((((((( • ))
               • )
               • )))
               • )
               • )

      (((((((( (((((((•)•)))•)•)) ))
               (((((((•)•)))•)•)) )
               (((((((•)•)))•)•)) )))
               (((((((•)•)))•)•)) )
               (((((((•)•)))•)•)) )

((((((((((((((((•))))•)•)•)))•)))•)))•)))


204x358
(((((((  • )
         • )))
         • )
         • ))

(((((((  (((((((((•))•)•)))•)•) )
         (((((((((•))•)•)))•)•) )))
         (((((((((•))•)•)))•)•) )
         (((((((((•))•)•)))•)•) ))

((((((((((((((((•))))•)•)•)))•)))•)))•)))
```

DECOMPOSITION VARIETIES

Standard binary decomposition
(compositional)
(polynomial)

Boundary decomposition
(cumulative)
(factored)

5=

```
4   1     4  ((•))
2   0     4   ( )
1   1     5    •
```

```
          2^2 + 2^0
             101
```

```
5    ((•))•
4    ((•))
2     (•)
1      •
```

```
          2(2(1))+1
           ( (1)) 1
```

7=

```
4   1     4  ((•))
2   1     6   (•)
1   1     7    •
```

```
          2^2 + 2^1 + 2^0
             111
```

```
7    ((•)•)•
6    ((•)•)
3     (•)•
2     (•)
1      •
```

```
          2(2(1)+1)+1
           ( (1) 1) 1
```

12=

```
8   1    8  (((•)))
4   1   12   ((•))
2   0   12
1   1   12
```

```
     2^3 + 2^2
       1100
```

```
12   (((•)•))
6     ((•)•)
3      (•)•
2      (•)
1       •
```

```
       2(2(2(1)+1))
        ( ( (1) 1))
```

35=

```
32   1   32 (((((•)))))
16   0   32
 8   0   32
 4   0   32
 2   1   34     (•)
 1   1   35      •
```

```
     2^5 + 2^1 + 2^0
           100011
```

```
35   (((((•)))))•)•
34   (((((•)))))•)
17    ((((•))))•
16    ((((•))))
 8     (((•)))
 4      ((•))
 2       (•)
 1        •
```

```
       2( 2(2(2(2(1)))+1 )+1
        (  ( ( ( (1))) 1 ) 1
```

204=

```
                                        204    ((((((•)•)))•)•))
128   1   128 (((((((•)))))))           102     (((((•)•)))•)•)
 64   1   192  ((((((•))))))            51      (((((•)•)))•)•
 32   0   192                           50      (((((•)•)))•)
 16   0   192                           25       ((((•)•)))•
  8   1   200     (((•)))               24       ((((•)•)))
  4   1   204      ((•))                12        (((•)•))
  2   0   204                           6          ((•)•)
  1   0   204                           3           (•)•
                                        2           (•)
                                        1            •
```

```
      2^7 + 2^6 + 2^3 + 2^2              2(2(2(2(2(2(2(1)+1)))+1)+1))
            11001100                      ( ( ( ( ( ( (1) 1))) 1) 1))
```

358=

```
                                        358   (((((((•))•)•)))•)•)
256   1   256 ((((((((•))))))))         179    ((((((•))•)•)))•)•
128   0   256                           178    ((((((•))•)•)))•)
 64   1   320   ((((((•))))))           89     ((((((•))•)•)))•
 32   1   352    (((((•)))))            88     ((((((•))•)•)))
 16   0   352                           44      (((((•))•)•))
  8   0   352                           22       ((((•))•)•)
  4   1   356       ((•))               11        (((•))•)•
  2   1   358        (•)                10        (((•))•)
  1   0                                  5         ((•))•
                                         4         ((•))
                                         2          (•)
                                         1           •
```

```
  2^8 + 2^6 + 2^5 + 2^2 + 2^1             2(2(2(2(2(2(2(2(1))+1)+1)))+1)+1)
         101100110                         ( ( ( ( ( ( ( (1)) 1) 1))) 1) 1)
```

73032=

```
65536   1    65536   ((((((((((((((((•))))))))))))))))
32768   0    65536
16384   0    65536
 8192   0    65536
 4096   1    69632       ((((((((((((•))))))))))))
 2048   1    71680        ((((((((((•))))))))))
 1024   1    72704         ((((((((((•))))))))))
```

```
  512   0    72704
  256   1    72960          (((((((•)))))))
  128   0    72960
   64   1    73024            ((((((•))))))
   32   0    73024
   16   0    73024
    8   1    73032               (((•)))
    4   0    73032
    2   0    73032
    1   0    73032

        2^16 + 2^12 + 2^11 + 2^10 + 2^8 + 2^6 + 2^3
```

```
73032   (((((((((((((((•))))•)•)•))•))•))))•)))
36516    (((((((((((((((•))))•)•)•))•))•))))•))
18258     (((((((((((((•))))•)•)•))•))•))))•)
 9129      (((((((((((((•))))•)•)•))•))•))))•
 9128      (((((((((((((•))))•)•)•))•))•)))
 4564       (((((((((((•))))•)•)•))•))•))
 2282        (((((((((((•))))•)•)•))•))•)
 1141         (((((((((((•))))•)•)•))•))•
 1140         (((((((((((•))))•)•)•))•))
  570          (((((((((•))))•)•)•))•)
  285           (((((((((•))))•)•)•))•
  284           (((((((((•))))•)•)•))
  142            (((((((•))))•)•)•)
   71             (((((((•))))•)•)•
   70             (((((((•))))•)•)
   35              (((((•))))•)•
   34              (((((•))))•)
   17               ((((•))))•
   16               ((((•))))
    8                (((•)))
    4                 ((•))
    2                  (•)
    1                   •
```

```
2(2(2(2(2(2(2(2(2(2(2(2(2(2(2(2(2(2(2(1))))+1)+1)+1))+1))+1)))+1)))
 ( ( ( ( ( ( ( ( ( ( ( ( ( ( ( ( ( ( ( (1)))) 1) 1) 1)) 1)) 1))) 1)))
```

```
1^2 = 1
11^2 = 121
111^2 = 12321   ...

decimal:

      ((•)•)•  x  ((•)•)•

      ((    •    )    •    )    •
         ((•)•)•   ((•)•)•   ((•)•)•

      (( ((•)•)• ) ((•) •)• ) ((•) •)•
      (( ((•)•)•    (•) •)•    (•) •)•
      (( ((•)•       •)••       •)••)•

      ((((•)••)•••)••)•
```