Distinction is Sufficient

William Bricken[1]
March 2017

In his seminal book *Laws of Form*, George Spencer Brown laid the foundations of iconic mathematics. By recording mathematical concepts in spatial forms rather than in symbolic strings, he demonstrated that logic rests elegantly upon one idea, *distinction*, the cognitive construction of difference. This article describes in detail how symbolic logic is permeated with irrelevant structure such as ordering, grouping, sequential steps, counting and duality. Iconic rationality rests only upon the deletion of irrelevant differences.

Contents

Figures

1. Introduction

Spencer Brown's seminal work *Laws of Form* (*LoF*) presents an iconic algebra that can incidentally be interpreted as propositional logic. LoF launches us into a postsymbolic territory where spatial forms condense symbolic complexity, where there is no syntax/semantics barrier, where objects are united with processes, where absence is a primary conceptual tool, and where the viewing perspective of the reader is directly implicated within the form. When applied to logic, Spencer Brown's iconic arithmetic challenges a foundational assumption of Western thought, that rationality requires dualism.

The purpose of this article is to demonstrate in detail that LoF is *not isomorphic* to symbolic logic. It is formally and conceptually much simpler. Said another way, LoF shows that common logic is baroque, burdened by too many structural restrictions, too limited stepwise linearity, too much computational mechanism, and too narrow a perspective on cognition. Symbolic logic informs rational thought, but only at the cost of the structural maintenance of verbal and textual strings of symbols. LoF foreshadows an entirely new technique for understanding the structure of formal proof and rational thought. We need only the concept of distinction, which can be expressed by containment forms that implement a partial ordering. Deduction does not necessarily require the duality of truth and its negation, it does not require the concept of negation at all. Semantic existence is sufficient to identify forms that are `TRUE`, those that are `FALSE` can be relegated to nonexistence.

By comparing seven different conceptual and notational formal systems to LoF, this article traces in detail how one accustomed to symbolic thinking might misunderstand Spencer Brown's iconic forms. Iconic notation provides structural room for both breadth and depth of expression, leading to an economy of concepts. LoF itself incorporates *only one relation* (containment), fully expressing Boolean logic within an algebra consisting of one constant, one variable and one binary relation. Transcribing the iconic notation of LoF into symbolic string notation converts the fundamental concept of containment into careful positioning of a sequence of replicated labels. A common confusion is the belief that free replication of symbols imposes no conceptual costs. The ordering and grouping required to disambiguate strings of tokens, for example, are properties of sequences of operations defined by the distributive axioms of logic and arithmetic. They can also can be understood as *incidental* to meaning, a property of the system of representation rather than of the things represented. Symbolic notation *imposes* sequence, suppressing the inherent parallelism of containment structures. Given sufficient processors we can access any number of containers all at the same time, but we cannot read a page of words all at the same time. Text incorporates the background white space of the page as a container of characters and words, however the space of the page provides only maintenance of textual sequence and is bejeweled with implicit conventions that allow us to organize strings of characters within an empty space. Icons in contrast are images that use space to convey meaning. Whereas symbolic logic is the lyrics of a song, iconic logic is the melody.

Unfortunately members of the formal community who have examined LoF are split into at least two factions. The antiquarian faction insists that LoF is just another syntax for common logic, that it is a unique notation for the same ideas that have been established over two millennia. The antiquarians endorse an understandably conservative perspective, that the logic that we already know is the ground beneath Spencer Brown's innovation. The representational slight-of-hand that changes LoF into classical logic is to add superfluous concept and structure that is not in *Laws of Form*. The postsymbolist faction sees distinction not only as the essence of logic, but also as the fundament from which logic and perception blossom. **Distinction** identifies a difference between content and context. Boolean algebra rests upon the two grounds of `0` and `1` (alternatively `TRUE` and `FALSE`); in LoF there is only One. Zero, nothing, *does not exist*. There is only *one* difference between `0` and `1`, that of change. *Difference alone*, as described by Spencer Brown and by Bateson, is a sufficient conceptual basis for rational thought.

Sections 1 and 2 identify the essential features of LoF that make it iconic rather than symbolic. Section 3 examines how these features redefine propositional logic, and then in Section 4 we construct an annotated notation that allows the chasm between iconic and symbolic form to be traversed. Section 5 discusses how the methods of predicate calculus bridge the chasm by the construction of incidental relational structure. These same mechanisms are presented in Section 6, embodied as conventional functions. Section 7 then examines an iconic version of LoF that supports parallel, asynchronous transformation of form. In Section 8 we present the algebraic axioms of LoF expressed in both iconic and symbolic formal structures, ending in Section 9 with a single variable iconic calculus that makes it evident that LoF does not even include the concept of a binary relation. There's a brief summary in Section 10, followed by an Appendix that compares examples of iconic and symbolic algebraic proof. The map from propositional logic to LoF is Figure 12 in the Appendix.

1.1 Exploration

George Spencer Brown ignited pockets of excitement, work and reflection across the world. His innovations in **postsymbolic logic** and distinction-based calculi have fueled a deeper philosophical understanding of the structure of reality, an insightful investigation into the nature of cognition, a clearer conceptualization of cybernetic formalism, and a broader perspective on mathematical and computational processes. As is widely known, Spencer Brown also ignited controversy, something that invariably accompanies innovation, especially in mathematics. The most dominant technical debate is whether *Laws of Form* is simply another notation for propositional logic, or whether LoF provides a completely different postsymbolic foundation for formal thinking. Both perspectives are correct, however the latter subsumes the former.

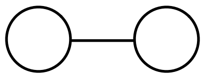
I've spent over three decades implementing Spencer Brown's concepts in symbolic software and in silicon hardware. If LoF is isomorphic to some existing and well-known mathematical system, then personally I'd find it to be not all that interesting. If LoF is a precursor to logic and to rationality, then surely we should be able to learn something by applying this calculus to some of the most complex applied logic problems known, the design and optimization of silicon circuitry composed of millions of logic gates. This article is not about massive optimization problems, it is about an ongoing attempt to find the bridge between LoF, conventional mathematics and the computational techniques used in the design of both semiconductor networks and the programming languages that instruct their behavior. The narrower goal is to specifically identify the symbolic mechanisms required to express LoF, and then to ask which of those mechanisms are inherent within LoF and which are essentially *invasive species*.

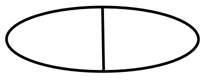
LoF is **iconic**, its forms look like what they convey. Conventional math, until very recently, is **symbolic**.[2] Its expressions are formally disconnected from what they mean. What *cognitive limitations* are imposed by the choice of a particular style of representation? What are the costs of attempting to separate the content of a communication from the method of its transmission? Does formulating our criteria for clarity, proof, and efficiency in terms of linear strings of symbols limit our capabilities for understanding the concepts that these strings supposedly represent? Could it be that what we consider to be rational and logical thought is burdened by approaches to communication that are too narrow?

Mathematics is an essential human endeavor that is deeply rooted in human cognition. Evolution of formal understanding is a natural process. Logic has evolved through the Aristotelian square of opposition to syllogisms to Boolean algebra to Gentzen natural deduction to modern computational approaches such as resolution, equational logic and category theory. Lakatos' thesis is that mathematical concepts evolve through a negotiated process of exclusion, reconceptualization, and generalization.[3] Symbolic logic rests upon duality, excluding the integrated unity of iconic distinction. Spencer Brown has provided a reconceptualization. Now is the time to think more generally, in terms of *unified systems rather than opposing factions,* in terms of webs of essential relations rather than discrete objects, in terms of ecological wholes rather than deconstructed pieces, and in terms of difference rather than truth.

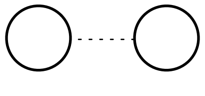
Item of Faith: *LoF is both formally and conceptually simpler than elementary logic.*

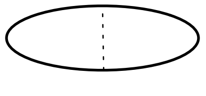
Formally simpler means that the language and the transformations of LoF map one-to-many onto the language and the axioms of propositional calculus. LoF reduces both the conceptional and the representational apparatus that supports our current foundation for formal thinking. We'll show that logic can be seen to consist of a single binary relation that does not require the infrastructure of sets or predicate calculus. Logic itself is much simpler than how we use it. LoF can help us to think logically without thinking symbolically. One key is to focus on the *relation between objects* rather than on the objects themselves.


BROADCAST


INTIMATE

The object-oriented broadcast model of communication presupposes two disconnected objects with a separate communication channel between them. The objects themselves embody difference. The intimate boundary model presumes two territories delineated by a common boundary that both separates and connects. Difference is embodied by the boundary. Eliminating the broadcast communication channel leaves two isolated objects. Eliminating the intimately shared boundary exposes a common unity.





LoF is a candidate for the *simplest elegant mathematical system*. If that is the case (or anything like the case), then we should be able to precisely trace a sequence of design decisions that connect the conceptual structure of LoF to that of symbolic logic and numerics. A theme is that the representation of formal thought is a **design choice**, one that strongly interacts with both content and cognition. Reading the book is not isomorphic to seeing the movie. The idea is to look for ways in which LoF simplifies and clarifies logic. Any structure that is not a straightforward isomorphic map from one system to the other is itself a formal difference between the two systems. We know the structure of both LoF and logic and we know the maps that connect the two (see Figure 12 in the Appendix), so one would expect that the design differences to be explored would be largely conceptual. There are indeed conceptual challenges, many arising from a diversity of ways that scholars read and interpret Spencer Brown's work. One conceptual difficulty is the tendency to define LoF in terms of classical symbolic mathematics, thereby obscuring its iconic form under a reading that imposes unnecessary restrictions.

There are also, however, many structural challenges. This is convenient since structural differences are overt, observable, and manipulable. These differences too can lead to interpretations that fail to acknowledge the conceptual and formal structure of LoF, that add non-existent structural elements to assure conformity, and that define a particular theory of representation as the only valid mode of communication. Because the symbolic approach is well-developed, it has little flexibility in reaching toward postsymbolic techniques. In comparison, restructuring iconic form to be symbolic is largely a matter of degradation, of obscuring the essential features of iconic form until both the representation and the intention are no longer visual or intuitive. Symbolic techniques support an ancient Greek belief that formal thinking is entirely cognitive, that sensation has no place within the rigor of mathematics. Iconic techniques support a postsymbolic perspective that cognition is *embodied*, that rigor arises from its biological and physiological context. A screenplay is expanded into a movie by enriching its sensual elements: sound, light, action. In reducing a movie to its readable text, sensuality reverts into the imagination, an enrichment of a different kind but one that weakens rigor by removing the validation of experience.

There is also a plethora of assumptions built into how we record formal systems on paper. These create difficulties whenever they are held to be invariant. We will focus on several. Classical mathematical perspective has embraced, for example, sequential processes at the cost of parallel concurrent processes. It has consecrated structural rearrangement at the cost of the simpler process of deletion. It has imbued the emptiness of the page we write upon with implicit structure such as an infinity of points. And it supports a belief that representation (syntax) is independent of meaning (semantics).

Software languages are a wonderful tool for locating design decisions within a structural system of representation. A software implementation environment is necessary to manage the diversity of exploration paths and for assuring the integrity of any bridges being built. Silicon computation reminds us, with insistence, that any computational mathematics is reducible to a string of `0`s and `1`s, which in turn is reducible to timed electrical currents either present or absent in each wire of a vast matrix of connected and buffered transistors. Computation protects us from imposing esoteric conceptual and cultural overtones and assumptions upon structural transformation. Conversely, structural necessity is an excellent mediator that helps to verify conceptual clarity.

1.2 Diversity of Languages

This article explores a diversity of representations for the `contains` relation, what Spencer Brown identifies as the definition of distinction.

Distinction is perfect continence.[4]

We'll compare seven different conceptual and representational systems to Spencer Brown's *Laws of Form*. Figure 1 identifies these languages and provides an example of the form of each. For the representation of containers, I'll be using `( )`, called *parens*, rather than Spencer Brown's ⏋, called *cross*, both for typographical convenience and for enhanced flexibility. And I'll use

the finger of interpretation ☞

to indicate when we move from one conceptual system to another. The finger is more than a syntactic map, it is an instruction to think differently.

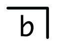
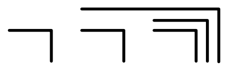
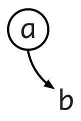
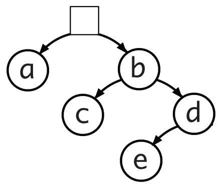
TYPE/NOTATION	FORM	EXAMPLE
iconic		
LoF cross		
parens notation	`(b)`	`() (()(()))`
hybrid		
annotated parens	$(b)_a$	$[(ø)_a((ø)_c((ø)_e)_d)_b]$
symbolic		
propositional logic	`NOT b`	`T or (T and F)`
ordered pairs	`{(a,b)}`	`{(U,a),(U,b),(a,ø),(b,c),(b,d),(c,ø),(d,e),(e,ø)}`
`contains` *relation*	∀b∃a Sab	SUa & SUb & Saø & Sbc & Sbd & Scø & Sde & Seø
`PUT` *function*	`Pba`	`a[[c[[ed]b]]U]`
iconic		
distinction network		

Figure 1: Eight Notations for Containment

To express containment in both iconic and symbolic languages, we will call upon these visually different notations.

— *Spencer Brown cross* is the original iconic form of a distinction.

— *Parens notation* is a convenient typographical shorthand for two dimensional enclosures. It represents arrangements of LoF crosses.

— *Annotated parens notation* adds labels as a bridge to a representation of containment arrangements in predicate calculus.

— *Propositional logic* is an interpretation of parens forms.

— *Sets of ordered pairs* are derived directly from the annotated parens notation.

— The *predicate calculus* description consists of quantified conjunctions of relational formulas. There is only one relation other than equality, the `contains` relation.

— The `PUT` *function* provides a functional representation of the `contains` relation.

— *Distinction networks* are an iconic notation that provides an alternative visualization of both parens and relational notations. *Dnets* use a single binary relation (`contains`) to connect nodes within a directed acyclic graph.

Figure 2 shows the relationships between the languages we will consider. The primary result of this exploration is buried deep within the techniques used to specify compound containment relations. Symbolic techniques for representing structures with both breadth and depth necessarily introduce *notational artifacts* that convert iconic LoF into symbolic logic. These incidental symbolic concepts include belief that absence must have a symbolic representation, that depth and breadth can be notationally equivalent, that labels can be freely replicated, that axioms for ordering and grouping are relevant, and that container boundaries must be impermeable.[5]

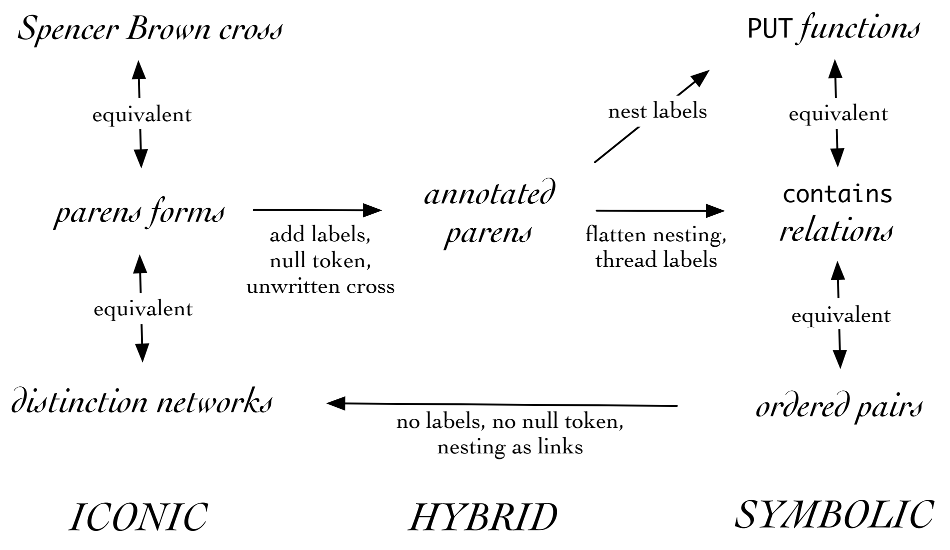


Figure 2: Relationship between Languages

2. What is Laws of Form?

A significant obstacle to this exploration is coming to an agreement upon what LoF actually means, on how it is intended to work, and on what the computational concepts built into the notation are. For example, LoF's use of void as an unwritten *state* must be conceptual, because physical computational devices and string-based software languages cannot use *absence* to propagate state information. Computation cannot incorporate our thoughts or our intentions unless they are explicit. The unwritten state is simply a crutch to reach out to classical two state thinking. The deep innovation within LoF is one-value thinking, a *unary logic*.

Here are five fundamental mathematical and philosophical Principles embodied in *Laws of Form* that we have implemented in computational systems. I'll suggest that these ideas are consistent with the LoF text, that they clarify what Spencer Brown was teaching us, and that they provide guidance about the conceptual and cognitive structure of LoF itself. The goal is quite narrow: to understand the explicit structural characteristics of Spencer Brown's calculus when they are expressed within the conceptual and notational structures of symbolic logic.

Principle of Void: *Void has no properties.*

This Principle is not explicit within LoF, perhaps due to Spencer Brown's wisdom, since the Principle itself might sound self-contradictory: *no properties* might be seen as a property. There are two voids. One we destroy by mentioning it. The other one we can refer to indirectly as the contents of an empty container. A **bounded void** is what remains after destroying the unmentionable void. All properties are properties of the boundary, and not properties of the contents, since there are no contents. By labelling and transforming only the *containers of nothing*, we can avoid any attribution of properties to absence itself. This principle permeates every implementation of LoF and is the fundamental difference between LoF and symbolic mathematical systems.

Principle of Existence: *Something is not nothing.*

The **initial distinction** is to posit a difference within void. The first distinction cleaves an empty space, converting nothing into something, destroying the emptiness to create a bounded void.

EXISTENCE $\neq (\ )$

In Existence the void is *not negated*, rather it is framed. A form comes into existence by a change in perspective. The change is to construct a boundary that draws attention. As Spencer Brown points out, the marked frame is also a label, a signal, an indication, an intention, a value, and an instruction. These concepts are at this point degenerate, different names for the same thing, concepts that are being called into being prematurely.

The Principle of Existence is sufficient to allow us to define what is meant by *distinction*, from both Spencer Brown's and Bateson's perspectives. A distinction-based perspective treats difference as fundamental. *Distinction is the cognitive act of constructing a difference where there is none.* A cognitive distinction can be represented by the difference between inside and outside of a parens, or the difference between under and not-under a cross, or the difference between a written mark and reading that mark.[6] The sides of a distinction define a choice of perspective. Difference is the act of crossing that boundary. At this point difference is not directional since we have yet to select which side of the distinction we are viewing from. Our choice of perspective defines what we mean by "outside". The characterization of a crossable boundary as a container incorporates an ecological perspective. When we are inside, as contents, the outer container defines our *environment*. As such it provides a comprehensive context and a limiting perspective. When we are outside, the container *object* shares our context. However, to view an object within our own context requires a further external environment, an outermost container for both our viewpoint and the object we are viewing.

Existence asserts that a container (a frame) and void are different, that inside and outside are different. The **difference sign** ≠ has both descriptive and constructive aspects. Statically the sign asserts that a container can be read as an object, as a **frame**. Dynamically the ≠ sign asserts that crossing a boundary changes what is visible. We can observe this change when crossing to the inside, when the former container-as-object becomes our containing environment. And we can observe the same change when crossing to the outside, when our surrounding environment becomes a discrete object. The ≠ sign itself appears to combine two logical concepts, equality and negation, however both of these concepts are derivative of what the ≠ sign means here. Classical mathematics focusses on identifying what is the same: *negation of sameness is difference*. Here difference is fundamental, indicating that outside differs experientially from inside. The iconic form of existence illustrates its meaning.

We can move closer to standard symbolic notation by introducing Spencer Brown's concept of an **unwritten cross**.[7] "An unwritten cross is common to every expression in the calculus of indications and so need not be written."[8] In order to locate LoF within symbolic systems, however, we will need to write what might be unwritten. Writing the unwritten cross introduces the idea of types of boundaries, some of which have the written property and some which do not. We'll use the expedient of making all unwritten boundaries explicit and differentiating them by a different boundary shape, the **shell bracket** 〔 〕. The unwritten cross serves only as an outermost container. It can be understood as belonging to the metalanguage, without semantic intent. We can now record Existence entirely from the outside perspective.

EXISTENCE 〔 〕 ≠ 〔 ( ) 〕

Comparing the two representations of Existence, we can see why Spencer Brown considered the unwritten cross to be unnecessary; it serves only to anchor our reading perspective. The unwritten container is not a semantic parens, it is a meaningless and therefore deletable distinction. We'll identify the unwritten container in symbolic notation by the token U.

Principle of Identity: *A distinction is itself.*

Identity defines the equal sign, and is an integral assumption of algebraic systems.

IDENTITY ( ) = ( )

What concepts and structures do we introduce with the equal sign? Why does it have two sides, and why does each side house identical forms? Where do the replicas come from? These structural questions do not require an excursion into what equivalence means. From living in physical reality we already know that equality must be an imaginary idea, constrained to cognition rather than to perception. Equality is a *transparent container,* one that does not indicate a difference. The two replicas on each side are the *same* form; identity helps us to believe that we can construct replicas without difference. Identity thus supports the illusion of a difference that does not make a difference. Making a distinction in void creates an inequality, a change from nothing to something. Failing to make a distinction creates an equality. The **initial equality** is our inability to be able to distinguish difference in absence.

NON-EXISTENCE =

We can use the unwritten cross to make this notation more appealing to a textual reading.

UNWRITTEN CROSS 〔 〕 = 〔 〕

Being able to identify identical replicas is an essential component of any transformation system. Operationally = means we can identify replicas in a potential cacophony of differences. It signals that the two forms it identifies can be freely exchanged. It is also a promise that any particular form will not turn into something else at the conceptual level. *Identity vows conceptual fidelity.* From the LoF perspective, = is an abbreviation that we will later be able to eliminate via void-equivalence. Identity marks a failure of perception, an inability to see difference. Identity turns into Equality when we assert axioms that identify different structures whose differences we choose, for some reason, to ignore. An assertion of equality identifies a voluntary blindness.

Principle of Containment: *There is only one relation, containment.*

One relation only is another way that LoF is not logic. "We have allowed only one kind of relation between crosses: continence… a cross is said to contain what is on its inside and not to contain what is not on its inside."[9] Unfortunately this definition includes a logical concept, `NOT`, as metalanguage. We will later eliminate it. The goal is to express an empty parens as a containment relation, without the support of logic, sets or numerics. Of course, *containment* is just a convenient and perhaps evocative label for a collection of properties that differentiate LoF forms from other structural relations such as equivalence, ordering, and the denizens of group theory. The semantic intent of containment is to identify distinction, or *difference*.

Containment is a relation between content and context, not between two content forms.

The direct structural implication is that we cannot be completely outside and still be able to observe anything. Observation is mediated by a common environment. With containment as the only relation, there is no objective perspective and there is no direct observation.

If there is only containment, how then can we interpret the Law of Calling, in which two parens appear to *not-contain* each other?

CALLING ( ) ( ) = ( )

It is a violation of the Principle of Void to assign a relation to juxtaposition of two forms, that is, to empower the void space between forms with an ability to bond forms together in a relation.[10] Forms within the same container are *independent* because the void within a container has no properties to support a mutual relation. One way to interpret Calling is that *not-contain* is not possible. Two apparent parens taken together without a mutual container is an illusion, there is only one. An alternative reading is that there is exactly one unique ground object `( )` so that replication can occur only in notation, as syntax, and not in meaning, as semantics. Multiple ground objects refer specifically to the single ground object. A third reading is that these two parens are indeed contained by an unwritten cross. The form `( )( )` necessarily implies an unwritten container, `[( )( )]` . From this perspective two crosses, each `NOT` contained by the other, are instead mutually contained by an outer container.

CALLING `[ ( ) ( ) ] = [ ( ) ]`

The unwritten container allows us to express Calling solely as containment relations. There is *only* containment of one form by another. Negation of containment is a meaningless concept within the LoF formalism, since every form is contained. Coming to terms with this reading is a central theme of this article.

Principle of Void-equivalence: *Void-equivalent forms are syntactically inert and semantically meaningless.*

If void has no properties, then how can we interpret the Law of Crossing, in which void appears to have a specific representation?

CROSSING `(( )) =`

The Principle of Void extends to explicitly recorded forms that are *defined to be void-equivalent* by rule and are thus equivalent to their absence. This is another fundamental difference between symbolic notation and LoF. In LoF some forms are *meaningless illusions*. These forms can be mentioned but are nonetheless useless. We'll call the form on the right-hand-side of Crossing a **double-parens**. An alternative perspective is that the double-parens of Crossing is indeed contained, by an unwritten cross. Although it is possible to call upon the Law of Crossing itself to assure an outer container, that type of design choice entangles transformation with representation.

CROSSING `[ (( )) ] = [ ]`

You may have noticed an extreme dependence upon both metalanguage and conventional concepts to describe the LoF Principles. Both Void and Existence seem to incorporate the logical idea of `NOT`.[11] What does *only one relation* mean in the face of replication? Calling can be read as denying the concept of accumulation while its apparent multiplicity at least invites the concept of counting. And then there is the imaginary quality of equality, a distinction that fails to distinguish. Eliminating these concepts is our major challenge.[12] The key is to be found in the Principle of Void-equivalence: *clarity is exposed through the deletion of meaningless form*.

3. Propositional Logic

Propositional logic is an *interpretation* of parens forms. Charles S. Peirce introduced iconic containment as a notation for logic (both propositional and predicate logic) at the turn of the twentieth century.[13] His Existential Graphs introduce many of the concepts incorporated into *Laws of Form*, with two major exceptions. Peirce's graphs are inferential, wedded purely to logical inference, while Spencer Brown's forms are equational, wedded to an algebraic perspective independent of the domain of propositions. The second difference is that Peirce maintains variables that stand in place of logical propositions, while Spencer Brown has developed an *arithmetic of form*, without variables and without propositions.

Propositional logic is a conceptual system resting upon the grounds of `TRUE` and `FALSE`. This duality creates the concept of negation. In the absence of duality, the map from logic to LoF is *many-to-one*. When propositional constants and connectives are transcribed into LoF forms, a diversity of apparently fundamental logical concepts evaporate.

```
TRUE
NOT FALSE
IF FALSE THEN FALSE          ☞    ( )
TRUE OR FALSE
FALSE NOR FALSE
```

The single empty parens cannot be said to *mean* the bevy of logical alternatives, since those logical alternatives classically define meaning itself. The validity and the value of our formal concepts are defined as being either `TRUE` or `FALSE`. The above logical expressions are syntactically equivalent but to be semantically equivalent logic would need to be conceptually redundant. ( ) shows us a single more fundamental concept that gives rise to the logical expressions that we perhaps indiscriminately associate with rational thought and understanding. Spencer Brown, Bateson, Varela, Heylighten and others have identified that single *ur-concept*: distinction. Spencer Brown's insight is that distinction is "a form of closure".[14] The cross marks a severance. Bateson's insight is that distinction is necessarily cognitive, that "difference is an idea."[15] Distinction has no physical presence and no metric, no place and no time.

> The explanatory world of *substance* can invoke no differences and no ideas but only forces and impacts. And, per contra, the world of *form* and communication invokes no things, forces, or impacts but only differences and ideas.[16]

Some logical expressions are so degenerate that they disappear upon transcription.

```
FALSE
FALSE OR FALSE               ☞
FALSE OR FALSE OR FALSE
```

These expressions fail to make a distinction, they can be *completely ignored without loss*. Yes, the concept `FALSE` is an unnecessary complexity that clouds rather than enlightens iconic thought. The

LoF arithmetic also includes an axiom that further extends the domain of meaningless concepts to meaningless but explicit *forms* that provide illusionary structure for meaningless concepts. These expressions too can be eliminated from rational cognitive processes without formal loss.

```
NOT TRUE
NOT NOT FALSE
IF TRUE THEN FALSE          ☞    (( )) =
NOT TRUE OR FALSE
NOT (TRUE OR FALSE)
```

The parens can be directly transcribed as logical NOR. Should we permit NOR to have any number of arguments, parens then expresses the entire diversity of the logical connectives. Here is a direct map from variary[17] NOR to LoF:

```
NOR[ ] = TRUE      ☞    ( )
NOR[a] = NOT[a]    ☞    (a)
NOR[a,b]           ☞    (a b)
NOR[a,b,c,…]       ☞    (a,b,c,…)
```

Classical logic imposes a requirement that connectives have exactly two arguments, without recognizing the fundamental idea of difference. Logical truth and negation and disjunction differ only in the *number* of the contents of a parens. Theoretically logic should not be built upon counting. From the LoF perspective, *how many* contents must be irrelevant since contents are mutually independent. Only containment matters. We call a container with no contents a *constant*; a container with a single content form, a *property*; and a container with two content forms, a *binary operation*. More than two content forms are called *n-ary operations*, with n being a count of the contents. In most computational languages, AND and OR, for example, are n-ary, but with the magnitude of n not being an essential factor. Computational AND, for example, terminates whenever the first FALSE argument is encountered, regardless of how many more arguments there may remain to be evaluated.

Symbolic logic is built upon the **duality** of TRUE and FALSE, it examines both sides of the coin separately. In contrast iconic logic is built upon the **unity** of distinction, it examines the coin as a whole. Taking sides is a choice of perspective, not a source of knowledge and truth.

4. Annotated Parens

To address computational and symbolic form, we will need to label different textual occurrences of parens, creating a hybrid notational system, *annotated parens*. Labels provide reference to both time and place. As emphasized by Spencer Brown, the cross is indeed a label, but only when viewed from a particular side, what we call *outside*. The cross allows us to know our place (we are not inside because we can see the label) and to know our time (by changing sides, we generate a before and after). Spencer Brown's cross rightfully carries no additional labels, however this is insufficient for bridging the gap between symbolic and postsymbolic concepts. In the presence of many boundaries we will need to know which boundary is which. Labelling provides multiple reference and is at the heart of the concept of abstraction. Yet it can be considered to be annotation, coming from the metalanguage and not an essential aspect of parens form.

With labelling, we can express the generic parens form of the `contains` relation.

container `a` `contains` *container* `b`	$(b)_a$

To map to symbolic notation, we will also need a label for void, a drastic departure from the intent of LoF forms. The problem can be mediated by using a token for void, ø, only in conjunction with an assertion that no other forms are contained by a container of ø. Rather than labelling void, then, ø is a token from the metalanguage that labels a constraint on the presence of contents.

container `a` `contains` *no forms*	$(ø)_a$
container `a` `contains` *no forms other than* `b`	$(b\ ø)_a$

4.1 The Annotated Arithmetic of Form

Labels provide sufficient mechanism to explicitly represent the LoF arithmetic within a symbolic language with one binary `contains` relation. Here are Crossing and Calling dressed up with computational labels and an unwritten cross.

CROSSING	`(( )) =`	☞	$[\,(ø\ (ø)_a)_b\,] = [\quad]$
CALLING	`( ) ( ) = ( )`	☞	$[\,(ø)_a\ (ø)_a\,] = [\,(ø)_a\,]$

We have acknowledged the absence of contents within container `a` by inserting ø. The token ø also identifies that the non-empty container `b` may contain no other forms if Crossing is to apply. The multiple occurrence of the label $(ø)_a$ in Calling acknowledges that the ground object is unique.[18] Both Laws need a void-equivalent outermost container, the unwritten cross. The outermost container for Crossing acknowledges the use of absence in the LoF notation, while the outermost container for Calling eliminates our propensity to assign a relational property to objects that share only nothing.

4.2 Incidental Structure

Our representation of the LoF axioms incorporates some implicit conventions that must be made explicit to a computational pattern-matcher. The symbolic description of a pattern-based transformation rule needs to identify four different types of structure.

— **explicit:** necessary structure that must be present
— **contextual:** structure that is contextually necessary but not changed
— **incidental:** structure that is incidental to the transformation, and not changed
— **forbidden:** structure that when present blocks a transformation

Explicit structure is explicitly present in a rule. *Contextual structure* is present in a rule but passes through a transformation unchanged. *Incidental structure* is not present in a rule and is passed through a transformation unchanged. *Forbidden structure* is present in a rule and causes a rule application to fail when it is found in the arrangement being transformed.

Applications of Crossing and Calling are independent of other forms within their context. This feature is a direct consequence of the Principle of Containment, that forms within any container are independent. To express this feature symbolically, we will need a generic pattern variable x_ to stand in place of any and all incidental structure that is irrelevant to the identification of patterns within rules. The underbar is an instruction to apply the label x to this superfluous structure, which may consist of zero, one or many independent forms. Thus every labelled parens will contain either ø to indicate "nothing else" or x_ to indicate "anything else." We now have

CROSSING (( )) = ☞ $[(ø\ (ø)_a)_b\ x\_] = [\qquad x\_]$

CALLING ( ) ( ) = ( ) ☞ $[(ø)_a\ (ø)_a\ x\_] = [(ø)_a\ x\_]$

To make the implicit conventions of string notation explicit, we've added these structural categories:

— **explicit:** existent typographical forms that are modified by rule
— **contextual:** typographical forms that are necessary but not modified by rule
— **incidental:** forms subsumed by the generic pattern variable x_
— **forbidden:** the presence of a ø token forbids other forms

One final notational elaboration is that transformations are insensitive to depth of nesting. Since the only type of form we have is a container, this elaboration allows us to apply the same transformation mechanism to every container regardless of depth of nesting. That is, *every container is a locally outermost container*. The outermost container can then be generic, identified by any labelled parens as well as by any unwritten cross. Again this is a direct consequence of the Principle of Containment: forms are independent regardless of their location *in breadth or in depth*.

CROSSING (( )) = ☞ $((ø\ (ø)_a)_b\ x\_)_d = (\qquad x\_)_d$

CALLING ( ) ( ) = ( ) ☞ $((ø)_a\ (ø)_a\ x\_)_d = ((ø)_a\ x\_)_d$

Symbolic conventions are ill-suited for specifying transversal of depth. Symbolic representation flattens the nesting relation between containers, changing the spatial correspondence between inside and outside into replicated labels in specific locations within a string expression. Distinction networks, a postsymbolic network notation introduced later, are much better adapted. For now, we'll simply define a self-similar operational rule for pattern-matching: *transformations apply to all containers concurrently*. The implementation of this rule, of course, will differ depending on hardware capabilities.[19]

Annotated parens notation is a bridge to the representation of containment forms in symbolic languages. Eliminate the annotations to reach LoF forms, incorporate the annotations to generate symbolic expressions. Since these annotations are necessary for string expressions, they identify formal differences between iconic and symbolic notations for LoF. String notations take the idea of labelling an object as a given freedom. Using the same label to identify different references to that object is also taken for granted. However, the Principle of Void for iconic notation suggests that labelling does have a cost, a label converts nothing into a boundary. The Principle of Existence asserts that a labelled (bounded) void is different than an unlabelled void. At a fundamental level, $(\ )_a$ is not the same as `( )`. Similarly `(ø)` is not the same as `( )`.

There is an analogy in symbolic string notation, the difference between a function symbol and an object symbol. Peano's definition of natural numbers includes the object `1` (which we will label here as $\texttt{obj}_1$) and the function $\texttt{next(obj}_n\texttt{)}$ which identifies the object that follows `n`. We have then

$$\texttt{obj}_2 = \texttt{next(obj}_1\texttt{)}$$
$$\texttt{obj}_3 = \texttt{next(obj}_2\texttt{)} = \texttt{next(next(obj}_1\texttt{))}$$

Symbolic systems label different numerical objects with different names (the cardinal numbers), but do not label different *applications* of the `next` function. We do not say

$$\texttt{obj}_3 = \texttt{next}_2\texttt{(next}_1\texttt{(obj}_1\texttt{))}$$

Section 5 that follows shows that replicating object labels is necessary to identify *relations* between objects. Relations are collected together by the symmetric operator `AND`, so that multiple relations are displayed in breadth (e.g. `R(a,b) AND R(b,c)`). The relation itself does not require unique labelling since an instance is labelled by its arguments. In Section 6 we'll see that replication of object labels is not necessary for *functions*, since function application itself is a method for generating labels.[20] Still we do not provide a unique label for each use of the function name. Functions however are composed together by asymmetric nesting in depth, for example, $\texttt{add1(times3(obj}_1\texttt{))}$. Symbolic notation has built into it a forced perspective that differentiates between objects and processes, between form and transform, enforced by a structural difference between breadth and depth and by an operational difference between symmetry and asymmetry. Iconic containment unifies this fragmented perspective, permitting a broader, more general conceptualization of formal description. The annotations added to each parens serve to separate the object aspect of parens from its operation aspect, thus undermining its essential unity. Both the nesting of function composition and the conjunctive composition of relations are notational choices that are incidental to logical meaning. These choices are aspects of linear text rather than inherent aspects of rational thought.

5. Restricted Predicate Calculus

Predicate calculus, also called **first order logic**, extends propositional logic with three concepts: domains of objects, quantification and relations between objects. We'll limit the domain of objects to containers only, and extend propositional logic by one relation only, `a contains b`, represented as a binary term `Sab`.

container `a contains` *container* `b` $(b)_a$ ☞ `Sab`

Both `a` and `b` are of the same type, but each is used in a different manner. It is as though we are mixing functions freely with relations, which of course we are since containers are both. A container acts like an **active operator** (it *encloses* its contents) and concurrently like a **relational object** (it is *enclosed* by its environment). The functional form of containment, `Pba`, is the binary function `b is-contained-by a`. We'll use the active voice and call this function `PUT`.

`PUT` *container* `b inside` *container* `a` $(b)_a$ ☞ `Pba`

5.1 Domain

Since all forms are arrangements of containers, containment is a global property of the domain of forms. We might approach a symbolic definition of the domain of LoF forms as a set of container labels extended by two special tokens and one relation.

let `C` *be a finite set of containers, labelled* `{a,b,c,...}`
and let `U` *and* `ø` *be special tokens,*
then the `contains` *relation* `S` *is on* `C union {U}` *to* `C union {ø}`
and is a subset of `C union {U} X C union {ø}`

A Cartesian product of two sets, `A X B`, is the set of all possible ordered pairs. We can identify a set `C+` consisting of all containers plus the special tokens `U` and `ø`. For `C+`, the `contains` relation is *on a set*. The `contains` relation associates two containers, both members of each pair come from the same set `C+`. The two special symbols then require that some members of the Cartesian product be excluded from valid containment relations. Later, several other constraints (the properties that define containment) will render other members of the Cartesian product to be not valid.

The familiar Cartesian product however is biased toward flattened string notations. We could unflatten the Cartesian promiscuity of *all possible pairs of label replicas* by considering the universe of containment pairs to be branches of rooted trees rather than a square matrix of pairs. LoF forms then are not strictly sets of relations, but rather belong to the graph category of directed acyclic networks.[21] In this non-Cartesian approach, we begin conceptually with a postsymbolic domain of trees rather than building the nodes and links of trees from symbolic structures such as the Cartesian set of all ordered pairs. Trees, like containers, represent their own structure when recorded in two and three dimensions. The omnipresent unwritten cross is the universal root. Replicated labels, when reunited as a single node with multiple links, convert trees into networks. Later, Figure 7 shows the constraints of the `contains` relation in network form. The change of perspective is from isolated binary relations that are later composed as sets or as conjunctions to the possible sets of these binary relations, to *composable networks*. Symbolically then, the domain of `contains` is all well-formed (typographically balanced) parens forms. Geometrically, it is all possible configurations of enclosed circles. Mathematically, it is the set of rooted trees. Physically, it is all the ways that we can assemble physical containers, one inside another. The tokens `ø` and `U` are not within LoF but rather are the cost of degrading LoF into a symbolic notation.[22]

5.2 Forms

A *form* is a valid configuration of containers. It is tempting to define parens forms in terms of the conventional definition of well-formed parenthesis expressions:[23]

> `( )` *is a parenthesis expression.*
> *If* `x` *is a parenthesis expression, so is* `(x)`.
> *If* `x` *and* `y` *are parenthesis expressions, so is* `x y`.

This definition in turn is modeled after the standard definition of expressions in propositional logic,

> *Propositional labels are expressions.*
> *If* `P` *is an expression, so is* `¬P`.
> *If* `P` *and* `Q` *are expressions, so is* `P v Q`.

It is the last line of the definition of parenthesis expressions that is suspect, because it constructs a well-formed structure by an operation that does not involve construction by containment. Placing `x` and `y` side-by-side introduces a conceptually separate method of construction. In terms of propositional logic, negation and disjunction are different concepts. The definition of parentheses confounds two different operations (containment and juxtaposition), and therefore does not characterize the structure of a containment relation.

We'll first define arrangements of containers inductively. We then rely upon the annotated parens form to arrive at a compact definition.

	ANNOTATED PARENS
`( )` *is a form.*	`(ø)`$_a$
If `b` *is a form, so is* `(b)`.	`(b)`$_a$
If `b` *and* `x_` *are forms, so is* `(b x_)`.	`(b x_)`$_a$

The first line defines a ground container with no contents. The second line permits containers to be nested. The final line permits multiple contents. Of course, the pattern variable `x_` is not a form, it is an appeal to induction over any number of content forms. Induction is also presumed by each of the above types of symbolic definition.

The problem introduced by the symbolic approach is to artificially differentiate breadth from depth, when both are simply consequences of containment. A compact definition of containment forms solely in terms of containment might look like this:

> `(x_)` *is a form, where* `x_` *stands in place of any number of other forms, including zero.*

For now we'll continue to explore conventional symbolic descriptions of arrangements of containers, with the intention of showing where the assumed bridge between symbolic notation and iconic form breaks down.

5.3 Ordered Pairs

An **ordered pair** is two labels arranged in order, with each position of the pair identifying a specific object. Generally sets of ordered pairs can stand in place of any symbolic relation.[24] Here the ordered pair `(a,b)` identifies a *containment relation*, with the first label identifying a container and the second label identifying a contained form. Both `a` and `b` are from the same set of container labels, although `a` may have other contents and `b` might also have contents. By describing containment using ordered pairs, we are simply saying that containment is a binary relation.

$$(b)_a \quad ☞ \quad (a,b)$$

For an ordered pair `(x,y)` to represent a containment relation, the **existence constraint** on the domain is that it includes only objects `x` for which at least one `y` exists. All objects in the domain can be containers. Thus we have introduced the null label `ø` to express an empty container as a pair, `(a,ø)`. The existence constraint on the codomain is that it includes only objects `y` for which at least one `x` exists. This constraint is met since all forms are contained. The unwritten container `U` is necessary only when we consider multiple forms at the top level of a compound form.

DOMAIN	$\forall x\ \exists y\ \ (x,y)$	*requires* `ø` *in codomain*
CODOMAIN	$\forall y\ \exists x\ \ (x,y)$	*requires* `U` *in domain*

The structure of ordered pairs then shows containment as sets of pairs of labels. LoF forms become sets of pairs.

empty container	$(ø)_a$	☞	`{(a,ø)}`
single containment	$(b)_a$	☞	`{(a,b)}`
nested containment	$((c)_b)_a$	☞	`{(a,b),(b,c)}`
multiple containment	$(b\ c)_a$	☞	`{(a,b),(a,c)}`
unwritten container	`[ b c ]`	☞	`{(U,b),(U,c)}`

Three related mechanisms are necessary to represent containment as ordered pairs:

— **sets** as a collection device for pairs and for ordering
— **symbolic replication** to generate multiple copies of labels, and
— **specific locations** of replica labels to indicate structure.

Successively nested containers are represented by several ordered pairs collected together as a set. In the case of the example below, $((((e)_d)_c)_b)_a$, these pairs indicate levels of nesting by showing the same label in the second position and also in the first position of a different pair.

Containers with multiple contents are also represented by a set of ordered pairs, in this case with the same first label for each pair. In the example below, $(b\ c\ d\ e)_a$, the same first label identifies the common container.

NESTED CONTAINMENT $\quad ((((e)_d)_c)_b)_a \quad$ ☞ $\quad \{(a,b),(b,c),(c,d),(d,e)\}$

MULTIPLE CONTAINMENT $\quad (b\ c\ d\ e)_a \quad$ ☞ $\quad \{(a,b),(a,c),(a,d),(a,e)\}$

Both nesting and multiple containment are expressed by *sets* of ordered pairs. More subtly, free replication of labels allows us to construct sets of ordered pairs that are internally strung together to represent relationship in breadth and in depth. Threading of freely replicated labels is how symbolic notation conveys and degrades the iconic structure of containment. When we get to iconic distinction networks, we'll see that many symbolic properties do not make sense within a postsymbolic form. Dnets require none of the three mechanisms of ordered pairs mentioned above.

5.4 The Ordered Pair Arithmetic of Form

Calling and Crossing can each be expressed as a *change* in a multiset of ordered pairs. We can see that the arithmetic of form can be expressed as *deletions of ordered pairs*.

CALLING $\quad (\ )\ (\ ) = (\ ) \quad$ ☞ $\quad ((ø)_a\ (ø)_a\ x\_)_d = ((ø)_a\ x\_)_d$

☞ $\quad \{(d,a),(d,a),(a,ø),(a,ø)\}$

$\quad = \{(d,a), \qquad (a,ø) \qquad \}$

The rule specifies that replicated empty containers can be deleted. Here, the ordered pairs are stacked vertically to draw attention to the pairs that are deleted. The deleted pairs correspond to the form $(ø)_a$. The incidental structure $x\_$ is not included in the ordered pairs since incidental structure is independent of the Calling transformation. Occurrences of the ground object $(ø)_a$ do not have different labels because empty parens forms are identical. This then aligns Calling with the deletion and creation of replicas within a multiset of ordered pairs.[25]

Crossing gives permission to delete a double-container, or constructively, to add a double-container.

CROSSING $\quad ((\ )) = \quad$ ☞ $\quad ((ø\ (ø)_a)_b\ x\_)_d = (x\_)_d$

☞ $\quad \{(d,b),(b,a),(b,ø),(a,ø)\}$

$\quad = \{ \qquad \qquad \qquad \}$

5.5 The `Contains` Relation

The symbolic *flattened viewpoint* of a relation is that objects are singular (unique) and relations between objects define composite structures. In predicate relations the nesting of containers is distributed over replicas of labels within different ordered pairs, while the collection of pairs is patched together with conjunctions. In predicate functions multiple containers are represented by nested function calls, which are more fully described in Section 6.

	PARENS ☞	ANNOTATED ☞	RELATIONS ☞	FUNCTIONS
empty				
	( )	$(ø)_a$	Saø	Pøa
single				
	(b)	$(b)_a$	Sab	Pba
nested				
	(((d)))	$(((d)_c)_b)_a$	Sab & Sbc & Scd	PPPdcba
multiple				
	(b c d)	$(b\ c\ d)_a$	Sab & Sac & Sad	PdPcPba
unwritten				
	〔 b c 〕	$(b\ c)_u$	SUb & SUc	PbPcU

Figure 3: Crossing the Iconic/Symbolic Chasm

Figure 3 compares the diversity of forms of containment in parens notation to annotated, relational and functional notations. What stands out is that the *structural varieties* of containment that are illustrated naturally in parens require the addition of elementary logic (the connector `AND`) to compose relations. Apparently we cannot engage in a symbolic relational language without relying upon the connectives of propositional logic. This same issue arises with ordered pairs and the introduction of set membership to represent forms as collections of pairs. For our purposes, ordered pairs and binary relations are different notations for the some things. Both require threading of symbolic replicas to identify particular types of structure. We might just as well say

$$(a,b) \;=\!\mathit{def}\!=\; \text{Sab}$$

In contrast, function nesting provides a composition mechanism for the `PUT` function that does not require the replication of labels. Where relational notation threads labels, function notation differentiates types of structure by different positions within nested function calls. Multiple contents are nested as the second arguments of `PUT` functions, while contents nested in depth are in the position of the first argument. Temporal sequencing of nested functions replaces spatial sequencing of replicated labels. Here is the comparison:

	ANNOTATED	☞	*RELATIONS*	☞	*FUNCTIONS*
NESTED CONTAINMENT	$((c)_b)_a$		Sab & Sbc		Pc(Pba)
MULTIPLE CONTAINMENT	$(b\ c)_a$		Sab & Sac		P(Pcb)a

What is apparent here is that parens forms are both objects constructed by relations (seen from outside) and processes connected by functions (seen from inside).

5.6 Quantification

Quantification has deep roots, going back to the distinction between the concepts *some* and *all* made by Aristotle. **Universal quantification**, $\forall x$, applies to all members of a domain. **Existential quantification**, $\exists x$, applies to at least one.

UNIVERSAL QUANTIFICATION	$\forall x$	*Every* x *is a container*
EXISTENTIAL QUANTIFICATION	$\exists x$	*At least one* x *is a container*

The empty container could be defined in the same manner as the empty set { },

AXIOM OF EMPTY SET	$\{\ \}_a$	☞	$\exists a \forall b\ \neg(b \in a)$
EMPTY CONTAINER	$(\ )_a$	☞	$\exists a \forall b\ \neg Sba$

However this definition confuses set membership with containment while calling upon an interpretation of ¬ as *does not contain*. Since everything is contained by some container (and since the LoF axioms do not call upon negation, see Section 8), there is no occasion to assert non-containment. We will instead adopt the convention that a negated term indicates an *invalid* form. The negation sign in front of Sba then indicates that it is not possible for all containers b to contain the same container a, a reasonable constraint when one considers physical containers. As well we avoid the rather absurd consequence of set theory, that every set contains the empty set. The subtle difference in the use of negation, between declaring that a given form does not contain another given form and declaring that it is not possible for a given form to contain another given form, will not lead to confusion. The does-not-contain interpretation is never used, while the cannot-contain interpretation is necessary to define the meaning of containment.[26]

One consequence of requiring all forms to be containers is that a universal outermost container, U, becomes necessary. So that U meets the containment constraint, it must be excluded from the codomain of the contains relation.[27]

NOT CONTENTS	$\neg(U)_a$	☞	$\forall a\ \neg SaU$

The negation sign in front of the parens form $(U)_a$ and in front of the relation SaU indicates a containment structure that is not permitted within the conceptualization of containment. It does not indicate that container a *does not contain* U, although this is indeed TRUE. Denial of being contained applies only to an unwritten cross since it exists solely as a void-equivalent outermost container.

As it turns out, quantification is remarkably sensitive to sets without members. To further reconcile symbolic and iconic approaches, we have elected to construct a special null token, ø. This approach achieves the objective of making it possible to express the empty container by a valid contains term, Saø. The special token is defined, however, as not existing.[28] Of course, the *representation* "ø" exists, otherwise we would not see it. However, the token has no referent.

DEFINITION OF ø $\quad\quad$ $ø \ =def= \ \neg\exists a \ \ a = ø$

This definition unfortunately rests upon equality while the Principle of Existence defines absence as an inequality. We are in danger of confounding notation with meaning.

EXISTENCE $\quad\quad$ $(\ ) \neq$

Within predicate calculus quantification, there is following relation between existence and negation:

NOT EXISTS $\quad\quad$ $\neg\exists x \ Px \ = \ \forall x \ \neg Px$

The assertion of non-existence of a single object with a given property P is the same as asserting that all objects do not possess the property P. We then arrive an alternative definition of ø. which conforms to the Principle of Existence.

ALTERNATIVE DEFINITION OF ø $\quad\quad$ $ø \ =def= \ \forall a \ \ a \neq ø$

The sensitivity to sets without members is an artifact of a conceptualization that objects must be in sets in the first place. This in turn creates a necessity for manipulation of symbols (by rearrangement for example) rather than the more elegant deletion of unnecessary symbols. By choice, we will still employ ø within symbolic terms. Its purpose after all is to define an empty container using a predicate calculus contains relation.

EMPTY CONTAINER $\quad\quad$ $(ø)_a$ ☞ $\exists a \ Saø$

Although the empty container is now described by an atomic contains formula, the label ø has no referent. From an iconic perspective, the Principle of Void forbids us from labelling nothing. In any image, the object of focus is brought into contrast by negative space, that is, by space that is not the object. **Negative space** pervades an image, there is no particular locale that supports the "nothing" label and no incentive to label the *background* as an object. We are left with a conundrum, a non-thing ø that is not within the domain of containers yet is being used within a containment relation.

An alternative is to omit ø and construct Sa, a *ground object* $(\ )_a$. Sa is then no longer a relation, it turns into a property asserted upon an atom with no interior and thus no capability to contain. If we were to add something to its inside (a common process in LoF), then the *type of thing changes*, from a property to a relation. So much for consistency.

Another alternative is to call upon what in predicate calculus is known as the problem of vacuous truth. When ∀xPx is asserted, the assertion is true whenever the set of all x is empty. For example, *all cats are white* is a true proposition whenever there are no cats.

VACUOUS NOTHING $(\emptyset)_a$ *=def?=* $\exists a \forall z\ Saz\ \&\ z \in \{\ \}$

Here we are simply embracing the weakness of set theory in addressing that which is not. ø is also necessarily excluded from the domain of the contains relation, ø is not a container.

NOT A CONTAINER $\neg(a)_\emptyset$ ☞ $\forall a\ \neg S\emptyset a = \neg\exists a\ S\emptyset a$

This definition supports our convention of negation as invalid form. The assertion that there does not exist an a that is contained by ø however does not convey the intent that ø is not a container. Bottom line is both that we will need to be very careful about using symbolic logic to describe LoF forms, and that we will eventually need to abandon hopes that LoF can even be described within the tools of predicate calculus.

5.7 Properties of Containment

Relations have a diversity of properties. Different collections of properties identify the well-known types of relation. For example, a relation that is reflexive, symmetric and transitive is an **equivalence relation**. Properties of relations act just like axioms to define structural constraints on forms. These properties assert that some structures are invalid. The *physical containment relation* is irreflexive and asymmetric and neither transitive nor intransitive.

Figure 4 shows the relational constraints that define our intended interpretation of containment, expressed in both the annotated parens and relational notations. Later, Figure 7 shows these same relational constraints visualized as postsymbolic network structure.

CONSTRAINT	ANNOTATED	RELATIONS
containment	$(b)_a$	$\forall b \exists a\ Sab$
empty container	$(\emptyset)_a$	$\exists a\ Sa\emptyset$
not a container	$\neg(a)_\emptyset$	$\neg\exists a\ S\emptyset a$
not contents	$\neg(U)_a$	$\neg\exists a\ SaU$
irreflexive	$\neg(a)_a$	$\neg\exists a\ Saa$
asymmetric[29]	$(b)_a \rightarrow \neg(a)_b$	$\forall a \forall b\ \neg.Sab\ \&\ Sba$
physicality[30]	$\neg.(c)_a\ \&\ (c)_b$	$\forall a \forall b \forall c\ \neg.Sac\ \&\ Sbc$

Figure 4: Symbolic Constraints that Define Containment

The relational constraints that define valid symbolic terms for containment also describe the structural characteristics of iconic parens forms. That is, parens forms provide a semantics for the symbolic expressions. The containment constraint is fundamental, all forms are the contents of some container. This constraint eliminates side-by-side containers as valid forms. Forms, like symbolic expressions, are singular objects. Since any container can contain any other container, any pairing of container/contained is permissible except for two formal (and physically intuitive) restrictions: *irreflexivity* and *asymmetry*. The *physicality* constraint can be also taken as a concrete description of forms, constraining representation to the possible ways that physical containers can be composed.

Irreflexive and Asymmetric

No object can contain itself. This **irreflexive constraint** asserts that any ordered pair in the form `(a,a)` is not valid.

STRICT CONTAINMENT $\neg(a)_a$ ☞ $\neg\exists a\ Saa$

Although all containers can contain any other container, some specific containment relations are excluded. Locally, for specific forms, if `a contains b`, then it is not possible that `b contains a`,

NO CYCLIC CONTAINMENT $(b)_a \rightarrow \neg(a)_b$ ☞ $\forall a \forall b\ \neg.Sab\ \&\ Sba$

The asymmetric constraint forbids reentrant containment forms. The intention is to exclude cyclic chains of containment, a refinement to be addressed later by distinction networks. Asymmetric relations are necessarily irreflexive.

Physicality

In physical models, no object can be contained by two different containers, because overlap (intersection) is not physically possible when container boundaries cannot be breached. Symbolically, however, the label of an object can be replicated and placed in different containers, in violation of the no overlap restriction. Structures with replicated labels are necessarily imaginary. Replication of labels is a primary difference between symbolic and physical models of containment.

PHYSICAL CONTAINMENT $\neg.(c)_a\ \&\ (c)_b$ ☞ $\forall a \forall b \forall c\ \neg.Sac\ \&\ Sbc$

The physicality constraint is not an aspect of *perfect continence* within LoF, and is not enforced in the definitions that follow. However, it is of interest that the transformation rules of LoF precisely specify ways to eliminate (and to create) symbolic overlap of containers. For some strict definitions of physical containment, the deletion of a container also deletes its contents. In LoF, containers can be deleted while their contents remain. There is only one axiom, Involution, and only one structure, the double-parens, in which this occurs.

Transitivity and Intransitivity

To consider transitivity of containment, two types of containment need to be distinguished. Shallow containment is not transitive; if `a shallowcontains b` and `b shallowcontains c`, then it does not follow that `a shallowcontains c`. A person is nourished by broccoli, for example, and broccoli is nourished by fertilizer, but that does not mean that a person is nourished by fertilizer. A bottle may contain water and the water may contain salt, but the bottle does not contain the salt.

We'll call the transitive version of containment `deepcontains`, `Dab`. For example, if a shopping cart contains a box of apples, then the shopping cart `deepcontains` the apples, as well as `shallowcontains` the box they are in.

TRANSITIVE $\forall a \forall b \forall c\ Dab\ \&\ Dbc\ .\rightarrow Dac$

The ordered pair structure of containment that we have adopted, `Sab`, is `shallowcontains`. Consider the form $((c)_b)_a$. which we have encoded as $\{(a,b),(b,c)\}$. If we were to consider containment to be transitive, then the pair (a,c) must be the case. However, the set of ordered pairs $\{(a,b),(b,c),(a,c)\}$ describes the form $(c\,(c)_b)_a$.

Although the `shallowcontains` relation is not transitive, it is also not intransitive. The genealogical `isParentOf` relation is an example of an intransitive relation. If `a isParentOf b` and `b isParentOf c`, then it cannot be the case that `a isParentOf c`.

INTRANSITIVE $\forall a \forall b \forall c\ Rab\ \&\ Rbc\ .\rightarrow \neg Rac$

If we were to declare in the form above that the pair (a,c) is forbidden, then we could not construct the form $(c\,(c)_b)_a$. However, violations of intransitivity will also be violations of the physicality constraint, because a contradiction of intransitivity implies that there are at least two different containers that contain the same object.

Essentially LoF forms do not accommodate the concept of transitivity. Structural transitivity incorporates the idea of *transitive closure*, that missing relations can be provided by rule. Void-based LoF uses a completely different mechanism. The axiom of Pervasion, described in Section 8, asserts that boundaries are semitransparent to forms on the outside. Outside forms are everywhere present at deeper levels, their explicit presence is optional since deeper replicas are void-equivalent. Again we see that LoF forms possess an entirely different kind of structure than symbolic expressions.

6. PUT Functions

The contains relation can viewed as a construction, that of inserting or *putting* one container inside the boundary of another container. Expressing containment as a function permits access to the well-developed field of function theory. LoF then specifies a particular variety of functional structure, a variety that, as might be expected, is not widely-studied.[31] We'll abbreviate the PUT operation as Pab.

```
PUT a into b      ☞    b contains a
       Pab        ☞       Sba
```

Every form can be PUT into some other form, satisfying the **existence constraint** for functions. Since the result of putting one specific form into another generates a new unique form, the **uniqueness constraint** for functions is also satisfied.[32]

Relational description is anchored in *space*, such as in a **relational database**. Functional description is anchored in *time*, such as in a **computational algorithm**. The label Sba describes an object or circumstance. The label Pab describes an action, or transformation. Both incorporate the restriction of sequential unfolding, each limited to its own dimension of descriptive existence.[33] The parens boundary, on the other hand, integrates both space and time into the same form, boundary as *contained object* from the outside and boundary as *environmental process* on the inside. Unlike characters of text, boundaries are both relations that separate and functions that connect.

Containment forms are compositions of PUT function applications. **Function composition** is expressed in string form by *nested function invocations*. PUT converts the textual necessity of threaded relational labels into dynamic nesting of non-replicated function application labels.[34] Function nesting and conjunctive listing then align to provide both spatial and temporal expression of containment in string notation. For example, the replicated label c in the ordered pairs description below is no longer required since it has been replaced by two applications of PUT into container c.

$(a\ b)_c$ ☞ {(a,c),(b,c)}

☞ PUT a into (PUT b into c) = PaPbc

We can PUT a form into a container whether or not that container has contents. This subtle shift essentially eliminates the textual glue mechanisms, such as logical conjunction and set membership, that are necessary to express multiple contents symbolically. This shift in notational perspective also changes our conceptualization of a container object from an isolated boundary to a boundary together with its contents. The domain is no longer a collection of individual containers, it is a collection of forms. The PUT function exhibits *closure*. Applying PUT to elements of the domain of function compositions generates another function composition. The algebraic structure of PUT thus consists of the following

The empty container is the null operation, Pøa = Paø.

The domain A *inductively includes all constructible containment forms.*

The binary PUT *operation* Pab *is on a set, mapping* $A \times A \rightarrow A$.

At this point, PUT operations on parens forms constitute a *magma*, ⟨A,P⟩, a set A equipped with a single closed binary operation P on the set A. Magmas impose no other axioms on their algebraic structure. The definition of PUT can be applied to the construction of rooted trees. One special object U is the root for all forms. Composition consists of attaching an existent rooted tree to the current root by putting any form a into U, via the function application PaU. The PUT operation itself can be defined iconically as a transformation

$$\text{Pab} \quad =\!\mathit{def}\!= \quad (\text{x\_})_b \text{ ==> } (a\ \text{x\_})_b$$

The incidental structure notation x_ acknowledges that function application is independent of any prior contents of b. No assumptions are made about x_, including the possibility that a is already contained by b and including the possibility that b is empty. The origin of the form a is not identified. Functions track changes rather than states. The arrow, ==>, can be read as substitution.

6.1 PUT Notation

We could, of course, use an infix notation for the binary PUT operation, such as

$$(a)_b \qquad ☞ \qquad \text{Pab} \quad \textit{alternatively} \quad a \oplus b$$

However, there is a standard compact algebraic notation for multiple applications of the same function. A function is represented by concatenation of its arguments, without writing the function name itself. In the case of PUT, Pab is written simply as ab. Brackets can be added, when necessary, to disambiguate the order of operations. To specify which function application is intended to occur first, we'll use square brackets rather than the customary parentheses within the standard compact notation. For example

$$(a\ b)_c \qquad ☞ \qquad \text{PaPbc} \quad \textit{alternatively} \quad a[bc]$$

The functional form a[bc] reads: PUT b into container c and then PUT a into the result, effectively putting a into c. This compact convention is a clear example of a space/time tradeoff within textual notation. Each occurrence of the operator P identifies a stack with exactly two arguments that follow. Should another P occur, that entire stack is treated as one argument. When the P flags are removed, the ordering of arguments is lost and must be replaced by brackets that identify the order of operations. That is, stack ordering in time is replaced by group ordering in space. The central observation however is that iconic parens do not impose symbolic ordering markers and thus are fundamentally more expressive than textual expressions. The intent of this and the following subsection is to demonstrate the equivalence of relational and functional thinking via their shared property of being expressed in textual strings.

The challenge with any function application notation is that the instruction to apply does not resemble the result of application. In computation, this problem has been well studied for the substitution function. Substitute A for B in C might be abbreviated as subst(a,b,c), but the abbreviation does not resemble the form of the new expression c_{new}. In the case of PaPbc above, we begin with empty container c and successively modify it by adding new contents

$(\ )_c$	☞	c
$(b)_c$	☞	$Pbc = c_{new}$
$(a\ b)_c$	☞	$PaPbc = Pac_{new} = c_{newer}$

We can take the composition of operations PaPbc to be a representation of $(a\ b)_c$, however it would be necessary to replay the construction sequence to retrieve the resultant iconic form.

We will also need to specify explicitly that container c begins the construction empty via Pøc. The empty container ( ) could be considered to be a ground object, however for PUT with pattern matching variables, every container has an implicit "and other contents" structure, $(x\_)_a$. This then requires us to declare explicitly when a given container is empty. For that we can use ø. We'll also introduce a *skeleton abbreviation* E for an empty container, Pøc. Although every empty container is identical, we could fudge this notation as E_c to preemptively declare a label for the currently empty container c.

$(ø)_c$	☞	Pøc	*alternatively*	øc	*or*	E_c	*or*	E
$(a)_c$	☞	PaPøc	*alternatively*	a[øc]	*or*	aE_c	*or*	aE
$(a\ b)_c$	☞	PbPaPøc	*alternatively*	b[a[øc]]	*or*	$b[aE_c]$	*or*	b[aE]

When the skeleton notation is intended to track construction of an arithmetic form from void, subscripted labels can be omitted, returning the functional representation to the unlabeled parens format. The skeleton notation shows clearly that the two arithmetic rules are associative variants of the same sequence EEU, yet yield different results. In this notation we can see that the rules of the arithmetic are an explicit statement of non-associativity.

CROSSING	[(( ))] = [ ]	☞	[EE]U = U
CALLING	[( )( )] = [( )]	☞	E[EU] = EU

6.2 Construction of Functional Forms

Any particular form can be expressed as a series of PUT operations into labelled containers. The empty double-container (( )) expressed as PUT operations is

$((ø)_a\ ø)_b$	☞	PPøaPøb *or* PE_aE_b	*alternatively*	E_aE_b

We have paid a price for the freedom provided by pattern variables, since they make algebraic specification easy at the cost of a more complicated notation for arithmetic forms. For functional expressions, specifying transformation from an existing form to a new form is more elegant than constructing the new form from scratch since each iconic arithmetic form *is* a record of its own construction. For example, to PUT any form into *any* other form is expressed concisely as

$((x\_)_a\ y\_)_b$	☞	Pab	*alternatively*	ab

but to PUT any form into an *empty* container, the empty container needs to have a label in order to avoid ambiguity during later construction.

$((x\_)_a\ ø)_b$ ☞ PaPøb *or* PaE_b *alternatively* aE_b

Exactly two empty containers as the content of the same empty container can be expressed as

$((ø)_a\ (ø)_b\ ø)_c$ ☞ PPøaPPøbPøc *or* $PE_aPE_bE_c$ *alternatively* $E_a[E_bE_c]$

Figure 5 contains an example of a LoF arithmetic form first annotated and then expressed as a sequence of PUT construction operations. To suppress algebraic generality, the specific arithmetic form must be explicitly constructed from empty containers. That is, every container must be introduced as initially empty within the annotated parens form.

parens	(( ) (( ) (( ))))
annotated parens	$((ø)_f\ ((ø)_c\ ((ø)_a\ ø)_b\ ø)_d\ ø)_g$
PUT *functions*	PPPabPcdPfg *with all labels i:* i=Pøi
PUT *functions with* E	$PPPE_aE_bPE_cE_dPE_fE_g$
compact	$[[E_aE_b][E_cE_d]][E_fE_g]$
compact labels only	[[ab][cd]][fg] *with all labels i:* i=[øi]
skeleton E *only*[35]	[[EE][EE]][EE]

Figure 5: Functional Notations for Constructing Parens Forms

Forms with multiple contents can be constructed by different sequences of operations. The example form above might also be expressed as

PfPPcPPabdg *alternatively* $E_f[[E_c[[E_aE_b]E_d]]E_g]$ *or* E[[E[[EE]E]]E]

In general, the order in which multiple contents are PUT into the same container does not matter. This commutativity over sequence of operations converts spatial commutativity into temporal commutativity, and foreshadows the parallelism of the LoF transformation axioms in Section 7. Embedded within the above PUT description of the example is the entire set of ordered pairs that define the example as a contains relation, one P flag for each ordered pair.

6.3 Properties of the PUT Function

The common properties of functions include associativity, commutativity, identity, inverse, and idempotency. PUT is neither associative nor commutative, as illustrated by these examples.

NOT COMMUTATIVE $((x\_)_a\ y\_)_b \neq ((y\_)_b\ x\_)_a$ ☞ Pab ≠ Pba
alternatively ab ≠ ba

NOT ASSOCIATIVE $((a\ x\_)_b\ y\_)_c \neq (a\ (x\_)_b\ y\_)_c$ ☞ PPabc ≠ PaPbc
alternatively [ab]c ≠ a[bc]

The empty container is not a zero for PUT. ø is also not a zero, since it is a convenience label and not an object.

NO ZERO $((x\_)_a\ ø)_b \neq ((ø)_e\ x\_)_a \neq (x\_)_a$ ☞ PaE ≠ PEa ≠ a
alternatively aE ≠ Ea ≠ a

Without a zero, there is no concept of a group-theoretic inverse operation, however there is a computational function inverse, GET a out of b.

GaPab = b *but* PaGab ≠ b ≠ a

Since the same forms can be constructed by different sequences of PUT operations, the PUT function does exhibit compositional, or temporal, invariants. In particular, the order that objects are PUT into the same container does not matter. Independence of container contents is expressed as commutativity over successive function compositions.

TEMPORAL COMMUTATIVITY PaPbc = PbPac *alternatively* a[bc] = b[ac]

Temporal commutativity is operational parallelism. Sequence of compositions is irrelevant for forms that can be PUT into a container concurrently. However, proof using PUT functions is severely burdened by the necessary ordering of binary arguments implicit in temporal construction. PUT makes the cost of ordering of function arguments explicit: the advantages of parallelism are lost.[36]

6.4 The Functional Arithmetic of Form

We have accumulated several functional notations and their shorthand forms, including a skeleton notation that is useful for illustrating the construction of arithmetic forms from a single empty parens, E. Figure 6 provides a comparison of each of these notations.

	CALLING	CROSSING
parens	( ) ( ) = ( )	(( )) =
annotated	$((ø)_a(ø)_a)_u = ((ø)_a)_u$	$(((ø)_a\ ø)_b)_u = (\ )_u$
PUT *functions*	PPøaPPøaU = PPøaU	PPPøaPøbU = U
PUT *with* E	$PE_aPE_aU = PE_aU$	$PPE_aE_bU = U$
abbreviated	$E_a[E_aU] = E_aU$	$[E_aE_b]U = U$
abbreviated labels	a[aU] = aU	[ab]U = U
abstracted E *only*	E[EU] = EU	[EE]U = U

Figure 6: LoF Arithmetic as PUT *Functions*

7. Distinction Networks

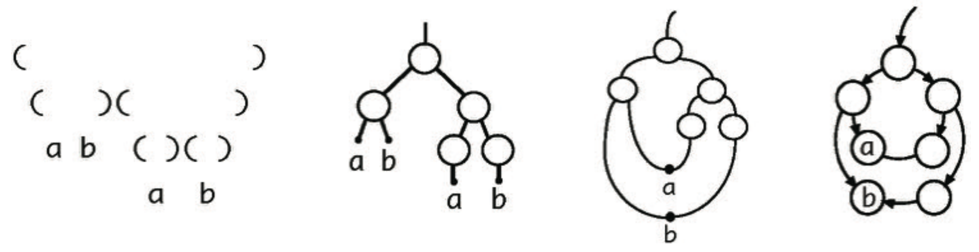
A distinction network (*∂net*) is a collection of **nodes**, or as we have been calling them here, containers, and a collection of **directional links** connecting pairs of nodes. The distinguished direction of connectivity could be called either `contains` or `is-contained-by`, depending upon our choice of perspective. Since a link is an ordered pair, we can consider a network to be *a set of ordered pairs of nodes*.

Dnets are a postsymbolic notation, a more natural iconic way to visualize containment. Basically dnets are **directed acyclic graphs with structure sharing**. By considering network structure to be a pictorial description without labelling the nodes, the presumed infrastructure of sets and predicate calculus is not necessary. For dnets, the link between two nodes can be seen as a shared boundary, while the nodes are the territories on either side of the boundary. The distinction boundary itself defines the difference between contents and context. The boundary provides context for its contents. Forms in the same context have the same boundary as their environment, however they have no relation to each other. This leads to an important advantage of modeling containment by network structures: networks highlight the potential for concurrent transformations.

7.1 Structure

A **form** is any sub-network of a given node. All nodes are roots for their contents, although some content nodes may have multiple roots due to structure sharing. **Acyclic networks** do not have loops, modeling the idea that two containers cannot mutually contain each other. Since all forms are contained the network is **connected**, the network analog of the concept of closure for string expressions. A node with no upper links is a global container, the **root** of a rooted graph. A node with no lower links is an empty container, a **leaf** of a rooted graph. Forms that do not share a common root are entirely independent of one another. In the network notation, downward exiting links have the same role as *pattern-variables*. They can connect to any network structure. An open-ended downward link acts like the symbolic **x_** indicating lower connectivity to at least one node. For algebraic accuracy, every node but leaf nodes should include an exit link.[37] A more casual notation is used below, to show shapes rather than formal specifications. In general the connectivity structure of the network is sufficient to identify a specific network uniquely.

Conversion of parens to network structure is shown below for the form `((a b)((a)(b)))`.



Parens are first extruded downward. Each parens is capped to construct a spatial enclosure, and direct nesting of parens is converted into network links. At this stage, a form is represented by a rooted tree. Structure sharing combines identical labels to build a network representation. The final circular display adds directional arrows and improves the appearance.

Here are some common examples of network structure.

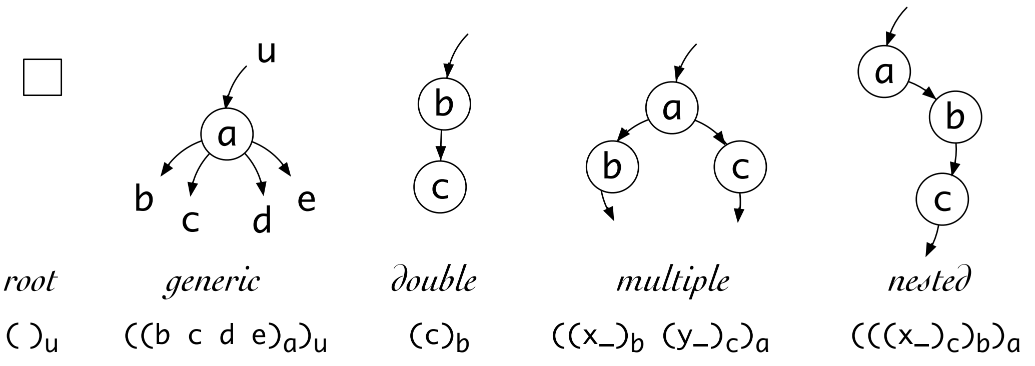


Figure 7 shows the symbolic properties of containment as structural properties of networks. The heavy bar across a link indicates that the link is not permitted. The bar serves the same role as negation, but in a visual format. Network constraints can be implemented without an infrastructure of sets and logic and numerics, each of which is an alternative language for abstracting the physical structure of a dnet. The change in perspective is from symbolic representation coupled with transformation by match-and-substitute to dynamic spatial activity choreographed by local message passing leading to local link deletion. Communication with direct neighbors achieves the same transformational objectives as match-and-substitute for strings, as inference for logic, and as union and intersection for sets. The foundations of our formal mathematics preserve conceptual and structural perspectives of a pre-computational era, and as such have no privileged status to describe inherently formal iconic computational processes. In particular, local concurrent message passing coupled with match-and-delete can be *described* by sets of ordered pairs and by conjunctions of relations, but the operations associated with sets and logic are not employed.

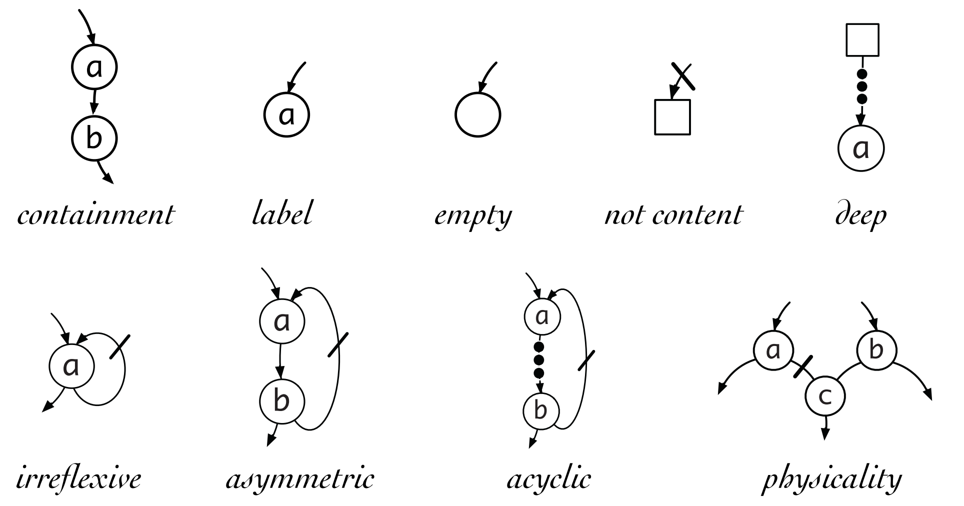
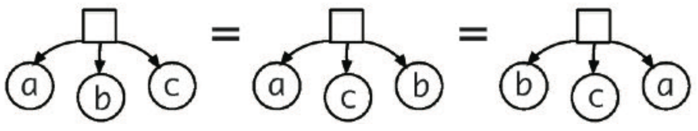


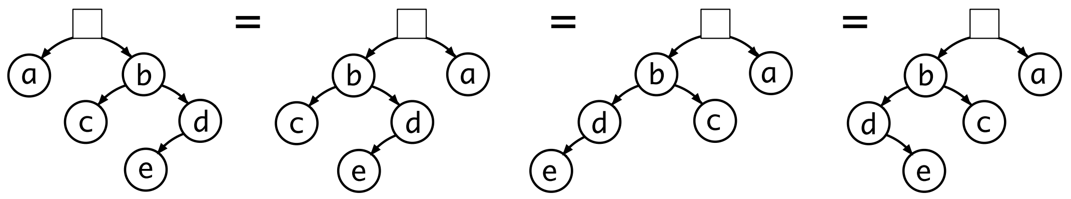
Figure 7: Iconic Constraints that Define Containment

7.2 Flexibility

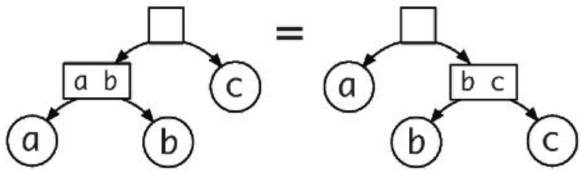
Rooted trees do not have a preferential ordering of their branches. Ordering of content nodes is not a concept within the iconic notation, there is no structural information within the collection of links to support such a notion. The visual notation of equal orderings below distorts the spatial perspective in favor of showing linear permutations to be equal. The iconic display, in contrast, is invariant under rotation in three-dimensions.



In the freedom of three dimensions, spatial transformation can be achieved by a greater diversity of methods, including two-dimensional spatial reflection, temporal network traversal, and three-dimensional rotation. By permitting the reader to inhabit the three-dimensional space of representation, rotation can be expressed equivalently as the movement of the viewpoint of the reader. Symbolic notation limits a reader's viewpoint, so relations such as commutativity must be expressed as a structural transformation of the representation. Iconic notation, in contrast, incorporates the concept of viewpoint, so that commutativity does not require restructuring the representation. Simply looking from a different perspective is sufficient.



An associative string function is also a temporal sequence of accumulation. In a network, this accumulation over time can be represented by nodes that create additional depth in the representation.



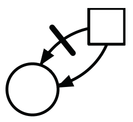
Spencer Brown's approach to associativity is similar to his approach to commutativity. From the perspective of iconic patterns, grouping the contents of a container (i.e. establishing an arity) violates the independence of those contents. Since containment itself is the iconic grouping operation, support of *binary* arity would require the insertion of additional containers, in effect the creation of two different types of container. Although depth and breadth are different structurally, the network itself unifies these apparently different properties of string representations. Conceptually, all nodes can be processed, traversed, and/or viewed at the same time. Both commutativity and associativity are *implementation details* rather than fundamental properties.

7.3 The Asynchronous Arithmetic of Form

From the perspective of communication, all nodes in a dnet are continuously active and responsive. From the perspective of computation, a dnet is a fine-grain massively parallel processor.[38] Dnet transformations are entirely local and exclusively limited to communication between a particular content form (a directly lower node) and the environment that contains it (a directly upper node). Messages flow in one direction only. Each node communicates only with its direct neighbors. There is *no global coordination*. That is, distinction networks act like cellular automata. In order to send messages, nodes have internal access to their upper and lower links, and can assess when there are no links. Messages instruct a neighbor to disconnect from the network. In the relational notation, these disconnections are represented as deletion of ordered pairs. In iconic notation, they implement a calculus in which network pruning is the primary operation.

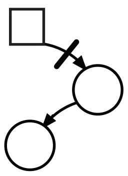
Crossing and calling can be expressed as dnets using a disconnection bar to indicate the structural changes specified by each rule.

CALLING



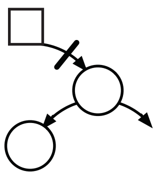
`( ) ( ) = ( )`

CROSSING



`(( )) =`

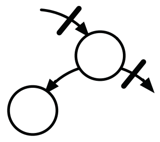
BOTH CONCURRENTLY



`(( ) x_) =`

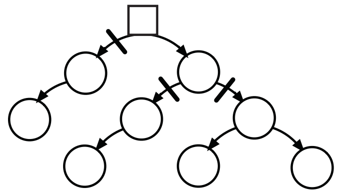
Although asynchronous message passing is designed for algebraic reduction, it applies in a simplified version to Crossing and Calling. As illustrated in Figure 8, both axioms condense into one message. All nodes concurrently examine their own connectivity. Nodes that have no lower connections send an asynchronous message to their upper connections, instructing each upper node to `disconnect` itself from the network by deleting all of its connections, regardless of connectivity. The root node of the network has no uppers to disconnect from. Effectively, the dnet implements an algebraic operation, deleting the pattern-variable `x_` that may be attached to the container of any leaf node. In Figure 8, the example dnet is reduced to a single node in one temporal step by three concurrent `disconnect` messages.

DISCONNECTION RULE



Condition: `lower links = 0`
Action: `send upper nodes DISCONNECT`

Condition: `receive DISCONNECT`
Action: `disconnect all links`



example `[(())((())(()()))] = [()]`

Figure 8: Asynchronous Crossing and Calling

8. Computation

Axiom systems guide the transformation of formal structure. These axioms are design choices about how we want to think. They impose distinctions upon formal thought by constructing partitions, by building boundaries that separate some concepts from others. It is the difference between the equivalence classes constructed by axioms that makes available different concepts. *Axioms themselves suppress difference.* Equality of form limits conceptual diversity.

Comparing the desirability of axiom systems requires not only a shared objective, but also a familiarity with how transformation is implemented. What makes a system of transformation axioms desirable? What motivates our choices to eliminate differences in form, to suppress variety? The discipline of mathematics is guided by *maximal abstraction*, by axioms that permit the most to be said with the least. Over a century ago, Poincaré offered this definition: "Mathematics is the art of giving the same name to different things."[39] What kind of axioms might achieve this goal? In *A New Kind of Science* Wolfram uses the programming language Mathematica to survey many thousands of potential axiom systems, concluding that there is nothing particularly fundamental about axiomatic logic.[40] Of the thousands of possible logic theorems, the ones that we as a culture have focused upon are not differentially interesting, at least not from a structural viewpoint.

One possible minimality metric for identifying the quality of an axiom system might be the total number of characters needed to express the set of axioms, but this has the severe disadvantage of rewarding tokens that are packed with hidden meaning. Counting the number of axioms is also a false measure, because by joining all the axioms by conjunction, we can always reduce the multitude to a single axiom that is usually extremely awkward to use. We then immediately apply self-transformations to that single axiom to free the more convenient versions as theorems.

The most elegant collection of concepts that underlie the representation of axioms might also be seen as desirable. Of course, humankind has yet to come to agreement on which concepts might have greater elegance. Historically, fidelity to physical observation served as grounding, however abstract logical and numerical systems are believed to be conceptual rather than concrete. There are also metamathematical constraints. Axioms should be independent of one another, consistent with one another, and complete in the sense that they can be used for what they are intended. Finally there is the psychological perspective, axioms should be very easy to understand, both in structure and in motive. Axioms should be *intuitive*. Mathematics has more or less ignored learning theory and the human perceptual system while identifying fascinating artifacts within the Platonic realm.[41] This finesse (or negligence if you choose) is not an option when working with an iconic notation.

Here is a LoF-like perspective. An axiomatic system is preferable when

— The concepts that underlie the representation are themselves elegant, succinct, widely applicable, mutually supportive, and transparently clear.
— The concepts map directly onto the representation in a sensual manner, so that seeing or reading or hearing or touching the representation elicits the concepts.
— The representation itself includes an extremely simple transformation system (i.e. proof theory) that suppresses long proofs while still generating interesting forms that stimulate new concepts and perceptions.

8.1 Pattern-Matching

Pattern-matching and substitution of equals is the primary algebraic mechanism for transformation of form, so we will need to identify just what such a mechanism requires and implies. **Match-and-substitute** has been foundational to mathematics since Leibniz. "Things are the same as each other, of which one can be substituted for the other without loss of truth."[42]

The capability to match forms is implicitly assumed in most mathematical systems, but it is far from trivial computationally. Substitution is usually hidden in the equal sign and in the cognitive deliberations of a supra-computational agent that we might call the Mathematician. From the algorithmic perspective, however, we must specify precisely what a pattern-matching engine should and can do. We can readily pass off the task of finding equal forms (i.e. using the equal sign) to the computer. Computers look for the same data structures in different parts of memory. *Structure sharing* is compiling identical structures into the same memory location. With a declarative programming language, we need only to decide upon which axioms (equalities) we wish to assert, the computer will take care of all the work to find those patterns and to apply substitutions. We do need to provide guidance about the *context* of symbolic substitutions for cases in which

— a specific transformation matches more than once
— the same transformation matches different parts of the same subform
— different transformations match parts of the same subform
— the result of a transformation again matches an available transformation
— a transformation might be beneficially applied in two different directions.

Symbolic transformation rules must include both positive and negative conditions for matching. **Positive conditions** specify the structures that must be present, **negative conditions** specify structures that must not be present. In contrast, the iconic description of both rules and forms uses empty space (i.e. non-representation) to specify negative conditions. Our final task then is to look at the diversity of results a computer might generate to find whatever we may be interested in. Often we use the computer to sort and filter its own results.

8.2 Iconic and Symbolic Calculi

Figure 9 shows the axioms of LoF in iconic parens notation. Spencer Brown identifies two axioms for the arithmetic and two axioms for the algebra.[43] His two axioms for the algebra of distinctions are historically grounded, raising parallels to Huntington's group-theoretic axioms in 1930s. Spencer Brown's Position and Transposition align with Huntington's Complement and Distribution. The figure also shows an alternative set of three axioms that are particularly convenient for computation. These three computational axioms are *void-based*, they implement transformation solely by construction and deletion of irrelevant structure. Spencer Brown's Transposition, in contrast, rearranges structure and depends upon matching pattern variables in non-nested locations.

initials of the arithmetic

CROSSING	`(( )) =`
CALLING	`( ) ( ) = ( )`

initials of the algebra

POSITION	`(a (a)) =`
TRANSPOSITION	`((a b)(a c)) = a ((b)(c))`

computational initials

DOMINION	`a ( ) = ( )`
INVOLUTION	`((a)) = a`
PERVASION	`a (a b) = a (b)`

Figure 9: The Iconic Transformation Rules of LoF

In the following displays, the LoF transformation rules are each presented for comparison in five different notations from previous sections: parens (introduced in Section 1.2), annotated parens (Section 4.1), sets of ordered pairs (Section 5.4), PUT functions (Section 6.4), and distinction networks (Section 7.3). As is the case for any equational system, rules can be applied in either direction, as construction or as deletion. The principle of void-equivalence suggests that structures that can be omitted should be omitted since void-equivalent forms do not contribute to necessary structure or to meaning.There is no concept of implication; deduction occurs through pattern-matching and substitution.

The ordered pair descriptions are stacked vertically to emphasize both the pairs that change and the explicit catalytic pairs that are necessary as contextual triggers but themselves do not change. Only the annotated parens form shows the incidental structure `x_` within each container. The variables in each pattern can be interpreted equally as labels standing in place of a single form or as pattern-variables standing in place of any number of forms. The former interpretation corresponds to a sequential implementation for which variables are operated upon one-at-a-time. The latter pattern-variable interpretation corresponds to a parallel implementation in which all forms within a particular container are operated upon concurrently. The pattern-variable interpretation leads to a more elegant set of axioms that are presented in Section 9. Finally the dnet representation uses the disconnection bar to indicate the available transformation.

Axioms of the Arithmetic of Distinction

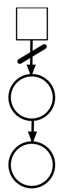
In each of the five systems of representation the axioms of the LoF arithmetic, **Crossing** and **Calling**, give permission to reduce forms via deletion to either a mark or to the absence of a mark.

CROSSING

parens `(( )) =`

distinction network

annotated $(((\emptyset)_a\ \emptyset)_b\ x\_)_u = (x\_)_u$



pairs

```
  {(U,b),(b,a),(b,ø),(a,ø)}
= {                       }
```

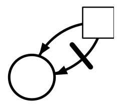
puts `[EE]U = U`

CALLING

parens `( ) ( ) = ( )`

distinction network

annotated $((\emptyset)_a\ (\emptyset)_a\ x\_)_u = ((\emptyset)_a\ x\_)_u$



pairs

```
  {(U,a),(U,a),(a,ø),(a,ø)}
= {(U,a),      (a,ø)      }
```

puts `E[EU] = EU`

Axioms of the Algebra

In Spencer Brown's axioms for the LoF algebra the container of the innermost replica in Position must otherwise be empty. Without this complete matching, Position takes the shape of the Pervasion, one of the computational rules below. Position is a special case of Pervasion for which the innermost `x_` is replaced by `ø`. The functional form shows that **Position** is a generalization of Crossing. Position strictly enforces the physicality constraint. In a void-based computational system, one way to assert a constraint is to make violation of that constraint void-equivalent. What began as an intuitive notion that objects cannot be contained by two different containers, is violated by a symbolic notation that permits free replication of labels, and is then corrected by LoF rules that assert these forms to be void-equivalent.

POSITION

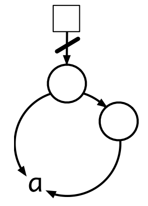
parens `(a (a)) =`

annotated $((a\ (a\ ø)_b\ z\_)_c\ x\_)_u = (x\_)_u$

pairs `{(U,c),(c,a),(c,b),(b,a),(b,ø)}`
`= {                              }`

puts `[aE][aE] = U`

distinction network



TRANSPOSITION

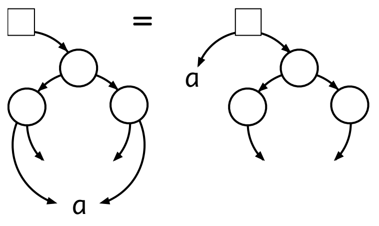
parens `((a b)(a c)) = a ((b)(c))`

annotated $(((a\ y\_)_f(a\ z\_)_g\ ø)_h\ x\_)_u = (a\ ((y\_)_f(z\_)_g\ ø)_h\ x\_)_u$

pairs `{      (h,f),(h,g),(h,ø),(f,a),(g,a)}`
`= {(U,a),(h,f),(h,g),(h,ø)              }`

puts `[c[aE]][[b[aE]]E] = [[cE][[bE]E]][aU]`

distinction network



Instead of deleting irrelevant structure, **Transposition** rearranges structure. In all notational styles, the descriptive complexity of rearrangement is apparent. Transposition is a *theorem* of the void-based computational axioms of LoF that follow. Many other generalizations of Transposition are available. For example there could be multiple forms containing `a` on the left-hand-side, leading to Broad Transposition.[44]

BROAD TRANSPOSITION `((a b)(a c)(a d)...) = a ((b)(c)(d)...)`

Computational Axioms

Dominion, Involution and Pervasion define a computational basis for the algebra of LoF. These three void-based rules simplify forms solely through deletion and construction of structure. In *Laws of Form*, Dominion is named Integration, Involution is Reflexion, and Pervasion is Generation. Different names are introduced here to emphasize that this basis is a different ground upon which to build the proof theory of the algebra of LoF. **Dominion** generalizes Calling while **Involution** generalizes Crossing. The computational axioms are therefore tightly connected to the LoF arithmetic. Each of the computational rules employs pattern-variables. The explicit form `a` is generalized to an incidental pattern-variable `a_` in Dominion and in Involution. Involution addresses double boundaries only, not the structure inside them. **Pervasion** formalizes the concept of *semipermeable boundaries*. It's pattern-variable `a_` is essential, to be matched and eliminated.

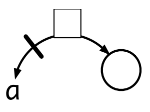
DOMINION

parens `a ( ) = ( )` *distinction network*

annotated $((\varnothing)_b\ a\_)_u = ((\varnothing)_b)_u$

pairs `{(U,a_),(U,b),(b,ø)}`
`= {       (U,b),(b,ø)}`

puts `a_[EU] = EU`



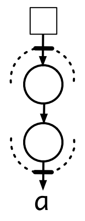
INVOLUTION

parens `((a)) = a` *distinction network*

annotated $(((a\_)_b\ \varnothing)_c\ x\_)_u = (x\_\ a\_)_u$

pairs `{(U,c),(c,ø),(c,b),(b,a_)}`
`= {                      (U,a_)}`

puts `[a_b]E = a_`



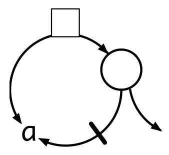
PERVASION

parens `a (a b) = a (b)` *distinction network*

annotated $(a\_\ (a\_\ y\_)_f\ x\_)_u = (a\_\ (y\_)_f\ x\_)_u$

pairs `{(U,a_),(U,f),(f,a_)}`
`= {(U,a_),(U,f)        }`

puts `[a_[bE]][a_U] = [bE][a_U]`



There is one remaining refinement to Pervasion. The pervaded replica `a_` in the original Shallow Pervasion is nested *one* level deeper than its outer matching pattern. The textual notation has not been adapted for transformation that crosses multiple depths. **Deep Pervasion** extends the depth of pattern matching to replicas at any deeper level of nesting. Succinctly, parens boundaries are *semipermeable* to replicas. We'll introduce a special type of boundary, `{ }`, to represent any depth (including zero depth), just like the ellipsis … represents any breadth. The *curly brace* `{ }` extends Pervasion to any depth of nesting. Similarly, the *link-ellipsis* in the dnet notation indicates any depth of network nesting, independent of intervening branches and boundaries. Since ordered pairs and PUTs are textual notations, neither accommodates a notation for arbitrary depth of nesting. `N` has been inserted into both of these notations as a default substitute for depth.

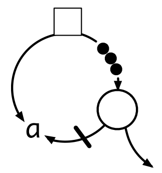
DEEP PERVASION

parens `a {a b} = a {b}` *distinction network*

annotated $(a\_\ \{a\_\ y\_\}_n\ x\_)_u = (a\_\ \{y\_\}_n\ x\_)_u$

pairs
```
  {(U,a_),(U,N),(N,a_)}
= {(U,a_),(U,N)       }
```

puts `[a_[bN]][a_U] = [bN][a_U]`



Many of the demonstrations in *Laws of Form* presume a sequential stepwise approach across depth of nesting. Spencer Brown demonstrated Deep Pervasion as a series of incremental steps, each step descending one level deeper into a form.[45]

```
a (  b (a c))
a (a b (a c))
a (a b (  c))
a (  b (  c))
```

This stepwise limitation is notational rather than conceptual. We can characterize Pervasion directly as a **deep rule** by asserting that *all boundaries are semipermeable* to forms on the outside. Shallower forms pervade all inward depths, including of course no depth.[46] A **semipermeable boundary** is the primary structural characteristic that distinguishes the interpretation of parens forms as logic.

9. Single Variable Calculus

The LoF computational algebra is a collection of three equations that assert void-equivalence of particular structural circumstances. In Section 8 these three rules are stated using variables to explicitly identify structure that is incidental to each rule. In a notation that makes incidental structure implicit, each rule can be reduced to address one variable only. The *one variable calculus* treats non-participating structure as truly non-participating, even in the notation. If a form is not transformed, it need not be recorded. From the iconic perspective, incidental structure is background and therefore does not need a label. To manage potential ambiguity, the null token is used to indicate a *necessary absence* of incidental, non-labelled content.

	SECTION 8		SECTION 9
IMPLICIT INCIDENTAL STRUCTURE	$(x\_)_a$	*is now*	$(\ )_a$
EXPLICIT ABSENCE OF STRUCTURE	$(\ )_a$	*is now*	$(ø)_a$

The computational axioms are shown in Figure 10 in the new single-variable notation. The figure includes three notational varieties for each axiom. The first variety is the original parens notation, the second variety is the annotated parens notation using the single variable convention of showing only necessary absence. The third equation shows this new notation without annotations.[47]

CROSSING		
	parens	`((( )   )  ) = (       )`
	annotated ø	$(((ø)_a\ ø)_b\ )_c = (\quad\ )_c$
	explicit ø	`(((ø)  ø) )  = (       )`
CALLING		
	parens	`(( )  ( )  )  = ( ( ) )`
	annotated ø	$((\ )_a\ (\ )_b\ )_c = (\ (\ )_a)_c$
	explicit ø	`((ø)  (ø)  )  = ( (ø) )`
DOMINION		
	parens	`(( )  A)  = (( )   )`
	annotated ø	$((ø)_a\ \ )_b = ((ø)_a\ ø)_b$
	explicit ø	`((ø)    )  = ((ø)  ø)`
INVOLUTION		
	parens	`(((A)   ) B)  = ( A  B )`
	annotated ø	$(((\ )_a\ ø)_b\ )_c = (\quad\ )_c$
	explicit ø	`((( )  ø)  )  = (      )`
DEEP PERVASION		
	parens	`(A  {A B} )  = (A  {  B} )`
	annotated ø	$(a\_\ \{a\_\ \}_b)_c = (a\_\ \{\quad\}_b)_c$
	explicit ø	`(a_ {a_ } )  = (a_ {   } )`

Figure 10: Axioms of the Single Variable Algebra

Dominion does not require specific variables. It asserts that when an outer parens contains an empty parens all other contents are void-equivalent, leaving a necessary null constant ø. Involution addresses only double parens, and also does not require specific variables. It declares all double parens to be void-equivalent. Deep Pervasion then is the only rule that requires matching a pattern-variable. It asserts that replicas of any form found nested at a deeper level are void-equivalent. Here, the form to be matched is identified by the pattern-variable a_.

Figure 11 shows the single variable calculus of indications in three notations: annotated parens, ordered pairs, and `PUT` functions. Here each notation is adapted for brevity to use implicit incidental structure. Any container that does not explicitly include a null token may contain any other contents. The variable `a_` in Deep Pervasion is a pattern-variable.

ANNOTATED	☞	ORDERED PAIRS	☞	PUTS
DOMINION				
$((\emptyset)_a\ \ )_b$		`{(b,a),(a,ø)      }`		`[øa]  b`
$=((\emptyset)_a\ \emptyset)_b$		`={(b,a),(a,ø),(b,ø)}`		`=[øa][øb]`
INVOLUTION				
$(((\ )_a\ \emptyset)_b\ )_c$		`{(c,b),(b,a),(b,ø)}`		`[a[øb]]c`
$=(\qquad\qquad)_c$		`={                 }`		`=      c`
DEEP PERVASION				
$(a\_\{a\_\}_n)_b$		`{(b,a_),(b,n),(n,a_)}`		`[a_n][a_c]`
$=(a\_\{\ \ \}_n)_b$		`={(b,a_),(b,n)       }`		`=   n [a_c]`

Figure 11: Notations for the Computational Axioms

The single variable calculus is extremely efficient to implement within a software pattern-matching language. Using the dnet representation, the three axioms can be implemented *in parallel*. The possible structural complexity of logic forms, of course, still remains. If this were not the case, then LoF would have solved the most important outstanding problem in computation, `P =?= NP`. The *satisfiability problem* is to determine whether or not an arbitrary expression in propositional calculus is a tautology. The question is: just how much effort should it take to identify a tautology, polynomial (`P`) or exponential time (`NP`)? With the single variable calculus, complexity resides solely in replicated tokens that occur in different contexts. The equation that most succinctly illustrates this problem is precisely Spencer Brown's Transposition axiom.[48]

Exponential effort can be characterized as making a guess about which transformation to apply, with the possibility that the guess may need to be changed, in the process requiring backtracking. Although Transposition is a theorem that can be demonstrated from the computational initials, deciding which forms to transpose can be exponentially complex. For example, it is possible that only one of the choices below will lead to easily determining if a more complicated form containing `(a b)(a c)(b c)` is a tautology.

```
(a b)(a c)(b c) ?=> (a ((b)(c))) (b c)
                ?=> (b ((a)(c))) (a c)
                ?=> (c ((a)(b))) (a b)
```

I have found no indication that LoF techniques can answer the satisfiability question. LoF offers a significant computational improvement in both space and time, but not a theoretical breakthrough. There is however a conceptual breakthrough, a new mental model for logic and for rationality.

10. Innovation

Since propositional logic can be expressed by each of the structural approaches considered herein, Spencer Brown has guided us to several different ways to think about and explore the logical foundations of mathematics and computation. What is clear is that simple logic, as expressed in symbolic form, is not that simple, nor is it elegant. It takes a thorough exploration to separate our symbolic heritage from Spencer Brown's iconic heresy, and to gain appreciation of the deeper contributions of Spencer Brown's innovations.

Logical proof, when expressed iconically, is successive and parallel deletion of void-equivalent forms. Complexity enters only as the efficiency of locating those deletions.

As is characteristic of the discipline of mathematics, Spencer Brown's work can be cast within the framework of existing mathematical tools, and it can also be seen to subsume those tools. The LoF axioms involve deletion and construction of images, rather than rearrangement of strings. Logical deduction currently rests upon accumulation of facts, rearrangement of collections of facts, and strategic planning to assemble and rearrange facts to arrive at conclusions. The techniques of modus ponens, disjunctive syllogism, reductio ad absurdum, etc. are ancient artifacts that still drive the organization of logical proof. None are particularly relevant to the direct iconic formulation of logic. Rational thought might instead be seen as selective forgetting of irrelevancies rather than as the accumulation and correlation of potentially relevant facts. Spencer Brown's point that associativity, commutativity, and arity are not central to the understanding of logic comes also with a more powerful frame of mind. It does not matter how many people share a room, nor is there any prerequisite ordering or grouping of those people. Here logical thinking is freed from linear structure of text and placed firmly within spatial visualization of image.

Symbolic expressions cannot do justice to iconic concepts. One insight is that structure, in both breadth and depth, should be apparent rather than abstracted. We have traced herein the various attempts to represent containment structure in symbolic terms. These include

— label replication,

— set membership,

— argument positioning,

— logical conjunction, and

— function nesting.

Only network linking appears satisfactory as a model of containment, primarily because networks, like LoF crosses, are iconic. We have not abandoned the algebraization of structure, LoF is algebraic. What Spencer Brown did was to introduce a new type of representation (spatial containment forms) into our well-known algebraic infrastructure, a change that has exposed many of the assumptions embedded in a string-based model of rigor. In the process he created a formal system that is not particularly group theoretic, but that is extremely relevant to computational processes. Although Spencer Brown's treatment of steps during a demonstration is pre-computational, somewhat inconsistent with his iconic cross notation, LoF forms are inherently both parallel and recursive.

The Appendix includes proof of a classical cornerstone of deductive reasoning, *modus ponens*, using four variants of the computational axioms of LoF: parens, ordered pairs, `PUT` functions, and distinction networks. Parens reduction uses match-and-substitute deletions on iconic containers to convert the LoF form of modus ponens into the form of `TRUE`. Ordered pairs and `PUT` functions use match-and-substitute on string representations. The dnet proof shows an iconic approach that includes asynchronous network pruning. All four proofs show the same transformation steps and all identify the matching process as well as the substitutions that are facilitated. Only the three computational axioms of LoF are employed, with no ancillary theorems.

We have not discussed many additional issues implicated by the conversion of an iconic representation of containment into symbolic text. Modeling features that are not sufficiently examined in this article include deep rules (rules that call upon the concept of pervasion), inherent parallelism, structure sharing, iconic proof techniques, void-equivalent catalytic forms, void-based programming languages, logic optimization, the impact of a unary value system upon classical logic, the reconstruction of the arithmetic of numbers based on Spencer Brown's and Kauffman's iconic techniques, and the many spatial and experiential languages and variants of parens forms. Reports exploring several of these topics are online at `http://iconicmath.com/`

My personal understanding of LoF has continuously evolved. I'm probably one of a few people who have had the privilege to be fully employed developing LoF applications for over two decades. When I look back over prior work I can see miscomprehension, oversimplification, projection, and struggle to step far enough away from what our culture teaches to be able to see LoF with fresh eyes. In particular my work has focussed on computational implementations, free of infinities, imaginary forms and deep philosophy. I've taken roads, often over years, that I currently see as misinterpretation of Spencer Brown's insights. There has always been an expanding frontier of discovery and exploration, replete with confusion, technical error and multiple revision, that has lead to a set of beliefs about Spencer Brown's work that are at first glance not explicitly delineated by Spencer Brown. A *central insight* is that our current mathematical foundations are far too complex. Rationality, at its core, is more about making distinctions and ignoring irrelevancy than it is about truth, conjunction, disjunction, negation, and implication. The LoF formalism provides a conceptual infrastructure for logic that does not include the binary choice of `TRUE` or `FALSE`, it does not include implication, and perhaps most surprising, it does not include the concept of negation. Value and relevance are determined by the context and the content of nested distinctions. Propositional logic incorporates a conceptual diversity thousands of years old that obscures what *Laws of Form* makes clear: *there is only difference.* Rational thought is not reliant upon dualistic choices such as true/false, good/bad, us/them and `1/0`. We can thank Spencer Brown for showing that logical clarity comes from knowing where we stand (within rather than under) while not making something out of nothing.

In summary, these are some of the sound-bites and bumper-stickers that have emerged.

ELEGANCE

— only one concept, distinction
— only one relation, containment
— only one value, existence
— void has no properties
— void-equivalent forms are illusions

STRUCTURE

— iconic not symbolic
— containers are environments
— we are within the form
— as above, so below
— object/process unity
— no syntax/semantics barrier
— independence of contents

PROCESS

— replication is not free
— deletion rather than rearrangement
— parallel as well as sequential
— no global coordination
— logic is semipermeable distinction
— implementation is a necessary ground
— counting isn't relevant.

Appendix: Proof of Modus Ponens

This appendix demonstrates the tautological structure of the logical inference rule of *modus ponens* using four notations: parens, ordered pairs, `PUT` functions, and dnets. Modus ponens is first transcribed into parens using the many-to-one map from logic to parens shown in Figure 12. The figure also shows an interpretation of the computational axioms as elementary logic.

BOOLEAN EXPRESSION	☞	PARENS FORM
`TRUE`		`( )`
`FALSE`		
`A`		`a`
`NOT A`		`(a)`
`A OR B`		`a b`
`A AND B`		`((a)(b))`
`IF A THEN B`		`(a) b`
`A NOR B`		`( a b )`

COMPUTATIONAL AXIOMS

`A OR TRUE = TRUE`	`a ( ) = ( )`
`NOT(NOT A) = A`	`((a)) = a`
`IF (A OR B) THEN A = IF B THEN A`	`a (a b) = a (b)`

Figure 12: The ☞ Map from Logic to Parens

When applying the computational axioms, tautologies reduce to an empty container which is interpreted as `TRUE`. Contradictions are deleted, relegated to nonexistence. Labels that remain after reduction support a semi-minimal logical form, as defined by an optimization criteria. Essentially, variables that remain are indeterminate in value. **Boolean minimization** requires several additional techniques that have not been discussed here. Absolute minimal forms are of course exponentially difficult to determine.

Here is modus ponens. Capital letters are used for logical variables and Quine dots for grouping.

MODUS PONENS `P and . P implies Q : implies Q` ☞ `(((P)((P) Q))) Q`

Modus ponens is quite general, it is intended to apply to simple propositions, to arbitrary formulas and sentences, and to arbitrary collections of formulas, all represented herein by pattern-variables as capital letters. The distinction between propositions, formulas, and sets of formulas is necessary within predicate calculus due to notational restrictions imposed upon the linear model of deduction. Pattern-variables are **generic**, combining the three levels of classical complexity of structure. From the perspective of classical logic, these articulations have been intimately involved in the definition and growth of logical structure and logical thought throughout the twentieth century. From the perspective of iconic structure, these articulations are notational artifacts.

Parens

The parens form of modus ponens is first reduced using the LoF computational axioms. Below, matching is indicated by a slash mark, so that `a/B` reads that pattern-variable `a` has been matched (technically, bound) to structure `B` within a form. Then an appropriate transformation rule from Figure 12 that incorporates the variable `a` is applied to generate a new reduced form.

```
(((P) ((P) Q))) Q       initial form
((     a     ))         match  a/(P)((P)Q)
  (P) ((P) Q)   Q             involution   ((a))=a
      (   a )   a       match  a/(P) Q, b/ø
  (P) (     )   Q             pervasion     a(a b)=a(b)
      (     )   a       match  a/(P) Q
      (     )                 dominion      a( )=( )
                        result ( )  ☞  TRUE
```

Ordered Pairs

The intention of the following proof using ordered pairs is to demonstrate that predicate calculus relations can be used for proof driven by LoF axioms as well as by Gentzen-type and by resolution inference systems. Ordered pairs that are deleted by rule application are marked by °. Empty forms are identified dynamically by failure to find other contents. The ordered pairs proof can be significantly more efficient if structure sharing is incorporated to strengthen what are essentially parallel operations.

ANNOTATED PARENS $\lceil\, (((P)_3\ ((P)_5\ Q)_4)_2)_1\ Q\, \rceil_u$

ORDERED PAIRS `{(u,1),(u,Q),(1,2),(2,3),(2,4),(3,P),(4,5),(4,Q),(5,P)}`

```
{(u,1),(u,Q),(1,2),(2,3),(2,4),(3,P),(4,5),(4,Q),(5,P)}         initial form
 (U,c)       (c,b) (b,a) (b,a)                         (c,ø)    match U/u,c/1,b/2,a/3 4
{  °   (u,Q),  °   (u,3),(u,4),(3,P),(4,5),(4,Q),(5,P)}  °               involution
       (U,a)             (U,b)             (b,a)                match U/u,a/Q,b/4
{      (u,Q),      (u,3),(u,4),(3,P),(4,5),  °   (5,P)}                  pervasion
                         (U,c)       (c,b)       (b,a) (c,ø)    match U/u,c/4,b/5,a/P
{      (u,Q),      (u,3),  °   (3,P),  °         (u,P)}  °               involution
                   (U,b)       (b,a)             (U,a)          match U/u,b/3,a/P
{      (u,Q),      (u,3),        °               (u,P)}                  pervasion
       (U,a)       (U,b)                         (U,a) (b,ø)    match U/u,b/3,a/P Q
{        °         (u,3),                           °  (3,ø)}            dominion
```

result $\lceil (ø)_3 \rceil$ ☞ TRUE

PUT Functions

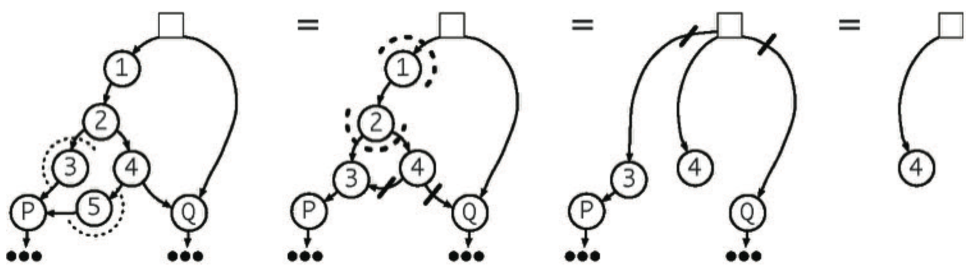
The next proof uses match-and-substitute steps on repeated function compositions. Since this mixes two essentially different styles (descriptive patterns and functional actions) the proof is both awkward and tricky. Pattern-matching requires rearrangement of function compositions, losing the advantage of non-replicated labels to simplify temporal commutativity. The PUT patterns for each axiom are shown to the right.

ANNOTATED PARENS	$[\ (((r)_3\ ((r)_5\ s)_4)_2\ ø)_1\ s]_u$
PUT FUNCTIONS	$PPPPrE_3PPPrE_5PsE_4E_2E_1PsE_u$

$PPPPrE_3PPPrE_5PsE_4E_2E_1PsE_u$	*initial form*	
$PPa \quad b\ E$	*match* $a/PrE_3PPPrE_5PsE_4,\ b/E_2,\ E/E_1$	
$P\ \ PrE_3PPPrE_5PsE_4 \quad PsE_u$	*involution*	PPabE=a
$P\ \ a\ \ PPa\ \ b\ \ c$	*match* $a/PrE_3,\ b/PsE_4,\ c/PsE_u$	
$P\ \ PrE_3P\ \ PsE_4\ \ PsE_u$	*pervasion*	PaPPabc=PaPbc
$P\ \ a\ \ Pbc$	*match* $a/PsE_4,\ b/s,\ c/E_u$	
$P\ \ PrE_3P\ \ s\ \ PPsE_4E_u$	*rearrange*	PaPbc=PbPac
$P\ \ a\ \ PPab\ c$	*match* $a/s,\ b/E_4,\ c/E_u$	
$P\ \ PrE_3P\ \ s\ \ P\ \ E_4E_u$	*pervasion*	PaPPabc=PaPbc
$P\ \ a\ \ P\ \ E\ c$	*match* $a/s,\ E/E_4,\ c/E_u$	
$P\ \ PrE_3\ \ P\ \ E_4E_u$	*dominion*	PaPEc=PEc
$P\ \ a\ \ P\ \ E\ c$	*match* $a/PrE_3,\ E/E_4,\ c/E_u$	
$P\ \ E_4E_u$	*dominion*	PaPEc=PEc
	result $[(ø)_4]$ ☞ TRUE	

Distinction Networks

The dnet parallel proof is included to illustrate multiple reductions occurring dynamically throughout the dnet. Nodes 3 and 5 first share structure, which then allows three concurrent deletions of connectivity (via Involution and Pervasion). Dominion then completes the proof. As with the other proofs, the remaining empty node is interpreted as TRUE.



Endnotes

1. **William Bricken:** Hi Folks. I'm best contacted at `william@iconicmath.com`. I work at Lake Washington Institute of Technology in Seattle as a professor of mathematics.

2. **Conventional math, until very recently, is symbolic:** Recently developed fields such as cellular automata, fractals, knot theory, and category theory incorporate visual iconic notations.

3. **a negotiated process of exclusion, reconceptualization, and generalization:** Lakatos (1976), *Proofs and Refutations.*

4. **Distinction is perfect continence:** Spencer Brown (1969), *Laws of Form*, p.1.

5. **container boundaries must be impermeable:** LoF distinctions, when interpreted for logic, are *semipermeable;* forms on the outside are arbitrarily also on the inside. See Section 8.

6. **the difference between a written mark and reading that mark:** The edge of this page is a distinction, the boundary between our physical reality and the representational images contained by the page. We default to viewing *notation* from the outside, by textual convention. The experience of being on the "inside" is that of not perceiving the frame, of transferring consciousness from awareness of our bodies to awareness of the meaning of the words we read. Immersive virtual reality is a recent technology that facilitates viewing symbolism from the inside. Perspective imposed by the Reader makes crossing from outside to inside different than crossing from inside to outside.

7. **Spencer Brown's concept of an unwritten cross:** *Laws of Form*, p.7.

8. **common to every expression in the calculus of indications and so need not be written:** *Laws of Form*, p.43.

9. **contain what is on its inside and not to contain what is not on its inside**: *Laws of Form*, p.6-7.

10. **an ability to bond forms together in a relation:** I made this error in much of my work from a couple of decades ago, calling mutual containment the `sharing` relation. It remains the dominant misconception in published commentary on *Laws of Form* today.

11. **Both Void and Existence seem to incorporate the logical idea of `NOT`:** Negation rests within the duality of classical logic. It requires us to take a side, exclusively one or the other. Within the form of the distinction, `( )`, we have access both to the outside and to the inside. LoF boundaries are *semipermeable*, what is inside is contained however what is outside can freely access the inside. Logical negation is impermeable, that which is `TRUE` does not necessarily have access to that which is `FALSE`.

12. **Eliminating these concepts is our major challenge:** This exploration is computational rather than philosophical, and does not incorporate the deeper ideas about void, existence, and identity explored by philosophers of mathematics such as Badiou and Rotman.

13. **notation for logic (both propositional and predicate logic) at the turn of the twentieth century:** For example, see Kauffman (2001), The Mathematics of Charles Sanders Peirce; Roberts (1973), *The Existential Graphs of Charles S. Peirce*; and Bricken (2006), Boundary Logic and Alpha Existential Graphs.

14. **Spencer Brown's insight is that distinction is "a form of closure":** *Laws of Form*, p.77.

15. **distinction is necessarily cognitive, that "difference is an idea":** Bateson (1972), *Steps to an Ecology of Mind,* p.481.

16. **no things, forces, or impacts but only differences and ideas:** *Steps to an Ecology of Mind,* p. 271. Emphasis in original

17. **Here is a direct map from variary:** A *variary* function applies to any number of arguments, including none. To accommodate this idea, standard function theory constructs a one argument function that takes a set of arguments as input.

18. **Calling acknowledges that the ground object is unique:** Analogously, we do not label multiple occurrences of a logical ground with different subscripts. We do not write $\mathtt{TRUE}_1$ `OR` $\mathtt{TRUE}_2$.

19. **implementation of this rule, of course, will differ depending on hardware capabilities:** Yes, there are significant questions about the possible interactions of rules. Bricken (1995), Distinction Networks.

20. **function application itself is a method for generating labels:** 2^{1000} is the *name* of a cardinal number that we would not care to identify explicitly. Even the familiar `1/2` (one half) is both the name of a fraction and the binary operation to achieve that fraction (divide `1` by `2`).

21. **belong to the graph category of directed acyclic networks:** A *directed acyclic graph* has no cycles, at least one source node with no inputs and at least one sink node with no outputs. A rooted graph has only one source node.

22. **the cost of degrading LoF into a symbolic notation:** These apparently benign extensions have contributed significantly to a general misunderstanding of LoF.

23. **the conventional definition of well-formed parenthesis expressions:** For example, Kleene (1952), *Introduction to Meta-mathematics*, p. 23.

24. **ordered pairs can stand in place of any symbolic relation:** The symbolic definition of an ordered pair specifies two sets, a singleton set to isolate the first label of a pair, and a doubleton set to isolate the second label. `(a,b)` =*def*= `{{a},{a,b}}`

25. **deletion and creation of replicas within a multiset of ordered pairs:** It is possible to eliminate the duplicates in Calling and thus avoid multisets by providing each empty container with a different label. However there is only one empty container, so that $(\varnothing)_a = (\varnothing)_b = E$. This is a case of the occurrence of different labels for the same form, similar to *Morning Star* and *Evening Star* which both identify the planet Venus. Once a form is inserted into an empty container, however, it becomes necessary to label that container.

26. **necessary to define the meaning of containment:** We are developing a non-dualistic version of logic that makes an interpretation of negation available when transcribing to symbolic expressions while negation itself is not conceptually necessary, having been relegated to nonexistence.

27. **excluded from the codomain of the `contains` relation:** For the interim, symbolic logic notation is mixed with boundary notation in order to trace the propagation of logic concepts across notations.

28. **The special token is defined, however, as not existing:** Unlike symbolic expressions, iconic forms such as the double-parens are meaningless and thus immediately deletable into non-existence. The idea of forms that are irrelevant and therefore expendable contributes significantly to efficiency of iconic computation.

29. ***asymmetric:*** Notice that we are eliminating the logical concept of implication in favor of the Boolean concept of conjunction.

30. ***physicality:*** The dot in the hybrid parens notation is Quine's notation for grouping, handy when parentheses have a different use. More dots mean greater scope.

31. **a variety that, as might be expected, is not widely-studied:** `PUT` however is an essential function within computational systems and computer languages. It is the instruction that puts data into a database. It's computational inverse is `GET`.

32. **the uniqueness constraint for functions is also satisfied:** There is a slight complexity. If a null token is repeatedly `PUT` into a container, replicas cannot accumulate: `PøPøa = Pøa = E`. The introduction of a token or representation for nothing inevitably leads to difficulties. It is not appropriate to assert `øø = ø` since absence in general cannot be replicated. Tokens that stand in place of absence undermine the semantics of their referent. Absence has no location, no cardinality, and no participation in or response to operations.

33. **each limited to its own dimension of descriptive existence:** The tradeoff between computational space and time is a dominant theme in theoretical computer science.

34. **into dynamic nesting of non-replicated function application labels:** It is not just coincidence that functions are represented by labelled boundaries.

35. ***skeleton*** `E` ***only:*** In Figure 5 the *skeleton* `E` *only* framework describes the construction of the example form in the following steps. The dynamic process descriptions are on the left and the static object descriptions are on the right.

```
                EE        ☞    (              ( ))
          [EE][EE]        ☞      ((( )) ( ))
      [[EE][EE]][EE]      ☞    ( ((( )) ( )) ( ))
```

36. **the advantages of parallelism are lost:** Within symbolic notation there is no free lunch. We must pay either replication of labels or temporal sequencing to compensate for a one-dimensional medium of expression.

37. **every node but leaf nodes should include an exit link:** Alternatively, exit links can be implicit and nodes that necessarily have no other exit links can be explicitly marked using the `ø` convention. This notational finesse is used in Section 9 to specify an elegant one variable calculus of form.

38. **a dnet is a fine-grain massively parallel processor:** Dnets have been used extensively to model and to optimize silicon circuitry. The first public demonstration of parallel logic computation using dnets was on a 16-node Intel Hypercube at IJCAI'87. Bricken & Gullichsen (1989), An Introduction to Boundary Logic with the Losp Deductive Engine.

39. **Mathematics is the art of giving the same name to different things:** Poincaré (1908), The Future of Mathematics Science and Méthode.

40. **there is nothing particularly fundamental about axiomatic logic:** Wolfram (2002), *A New Kind of Science*, pp. 714–846.

41. **while identifying fascinating artifacts within the Platonic realm:** I have adopted the modern perspective that math is a human activity, therefore it should at least reflect human physiology.

42. **which one can be substituted for the other without loss of truth:** Frege (1884), *The Foundations of Arithmetic*. p. 76. quoting Leibniz.

43. **two axioms for the arithmetic and two axioms for the algebra:** Spencer Brown uses the word *initials* to identify what we would conventionally call axioms.

44. **multiple forms containing `a` on the left-hand-side, leading to Broad Transposition:** *Laws of Form*, p.38.

45. **each step descending one level deeper into a form:** *Laws of Form*, pp. 39–40.

46. **forms pervade all inward depths, including of course no depth:** Pervasion has its roots in the alchemist's creed: *As above, so below*. Note that the background void pervades all forms. Pervasion is better visualized as pervasive presence of each form throughout inner depths of nesting, as if the parens inwardly facing boundaries are void-equivalent and do not exist.

When parens are interpreted as functions, Deep Pervasion reaches across function boundaries to change the arguments of nested function applications. E.g., `f[a,f[b,f[a,c]] = f[a,f[b,f[c]]]`. Incidentally, the semipermeability of distinction boundaries is a property only of the logical interpretation. When parens forms are used to axiomatize numbers, parens boundaries are impermeable.

47. **The third equation shows this new notation without annotations:** The *explicit* `ø` computational axioms cannot be transcribed into the notation of symbolic logic since implicit non-representation of incidental structure is not possible in symbolic forms.

48. **most succinctly illustrates this problem is precisely Spencer Brown's Transposition axiom:** Bricken (2002), Computational Complexity and Boundary Logic.

References

Badiou, A. (2008). *Number and Numbers.* Cambridge, UK:Polity.

Bateson, G. (1972). *Steps to an Ecology of Mind.* New York, NY:Ballantine.

Bateson, G. (1991). *A Sacred Unity.* New York, NY:HarperCollins.

Bricken, W. (1992). Spatial Representation of Elementary Algebra.*1992 IEEE Workshop on Visual Languages, Seattle, WA.* Los Alamitos, CA:IEEE Computer Society Press. p.56-62. Also available at `http://iconicmath.com/algebra/spatial`

Bricken, W. (1995). Distinction Networks. In I. Wachsmuth, C.R. Rollinger & W. Brauer (Eds.), *KI-95: Advances in Artificial Intelligence*. Berlin:Springer. p.35-48. Also available at `http://wbricken.com/htmls/01bm/0105arch/010501dnets.html`

Bricken, W. (2002). Computational Complexity and Boundary Logic. Available at `http://iconicmath.com/logic/boundary`

Bricken, W. (2006). Boundary Logic and Alpha Existential Graphs. Available at `http://iconicmath.com/logic/boundary`

Bricken, W. (2006). The Mathematics of Boundaries: A Beginning. In D. Barker-Plummer et al (Eds.). *Diagrams 2006, LNAI 4045,* Berlin:Springer. p.70-72. Also available at `http://wbricken.com/htmls/01bm/01-math.html`

Bricken, W. (2006). Syntactic Variety in Boundary Logic. In D. Barker-Plummer et al (Eds.). *Diagrams 2006, LNAI 4045,* Berlin:Springer. p.73-87. Also available at `http://wbricken.com/htmls/01bm/0103notate/010301nonsymb.html`

Bricken, W. & Gullichsen, E. (1989). An Introduction to Boundary Logic with the Losp Deductive Engine. *Future Computing Systems* 2(4). p.1-77.

Bricken, W. & Nelson, P. (1986). Pure LISP as a Network of Systems. Proceedings of the Second Kansas Conference: Knowledge-Based Software Development, Kansas State University.

Frege, G. (1884). *The Foundations of Arithmetic.* Evanston, IL:Northwestern University Press. Also in J. vanHeijenoort (Ed.) (1967) *From Frege to Godel*. Cambridge, MA:Harvard U. Press.

Heylighen, F. & Joslyn, C. (2001). Cybernetics and Second-Order Cybernetics. In R.A. Meyers (Ed.), *Encyclopedia of Physical Science & Technology* (3rd ed.). Oxford, UK:Elsevier.

James, J. & Bricken, W. (1992). A Boundary Notation for Visual Mathematics. *1992 IEEE Workshop on Visual Languages, Seattle, WA.* Los Alamitos, CA:IEEE Computer Society Press. p.267-269.

Kauffman, L. (2001). The Mathematics of Charles Sanders Peirce. *Cybernetics and Human Knowing*, 8(1-2), p.79-100.

Kleene, S.C. (1952). *Introduction to Meta-mathematics*. New York, NY:North-Holland.

Lakatos, I. (1976). *Proofs and Refutations: The Logic of Mathematical Discovery*. Cambridge, UK:Cambridge U. Press.

Poincaré, H. (1908). The Future of Mathematics Science and Méthode. Retrieved on 5/17 from `http://www-history.mcs.st-andrews.ac.uk/Extras/Poincare_Future.html`

Roberts, D. (1973). *The Existential Graphs of Charles S. Peirce*. Paris:Mouton.

Rotman, B. (1987). *Signifying Nothing: The Semiotics of Zero*. Palo Alto, CA:Stanford U. Press.

Spencer Brown, G. (1969). *Laws of Form.* London, UK:George Allen and Unwin.

Varela, F. (1979). *Principles of Biological Autonomy*. New York, NY:North Holland.

Varela, F. & Goguen, J. (1978). The arithmetic of closure. *Journal of Cybernetics,* **8**, p.291-324.

Wolfram, S. (2002). *A New Kind of Science*. Champaign, IL:Wolfram Media.