REASONING USING LOSP

William Bricken
June 1985

CONTENTS

INTRODUCTION

We propose a new representational formalism for logic that is compatible with the programming language LISP. This formalism, called Losp, is derived from the single operator logic of G. Spencer-Brown called the *Laws of Form* (LoF). For the mechanization of reasoning, Losp promises the distinct advantage of an elegant representation that eliminates many of the computational steps incumbent in traditional approaches. The technical discussion includes a description of the formalism and examples of its application.

TECHNICAL DISCUSSION

Propositional calculus is the primary mathematical tool for the implementation of automatic reasoning systems and semiconductor circuits. The mathematics of Losp is morphic with propositional calculus (PC). We will distinguish the mathematics conceived by Spencer-Brown (LoF) from the computer language Losp (Losp is a name, not an acronym, and is not composed of capital letters).

Several new representational techniques within LoF simplify the transformation of expressions and the proof of theorems:

1. *Equationality*: Equality of expressions in LoF is morphic with logical equivalence in PC, converting the inference rules in PC into algebraic transformation rules in LoF.

2. *Abstraction*: The basic connectives in LoF are inherently associative, commutative and n-ary. Transformations involving these properties are unnecessary.

3. *Unified representation of function and description*: A LoF expression is both a descriptive state and a functional form. The interpretation placed upon the representational context determines whether the expression is dynamic or static. This convention facilitates an ease of transfer between syntactic substitution and computational simplification.

4. *Implicit duals*: Only half of each duality (e.g. TRUE/FALSE, AND/OR, ALL/EXISTS) is represented explicitly. The other half is unlabeled, implicit in the representational context. The resulting economy of representation permits rapid simplification of expressions.

5. *Single token representation*: One single explicit token is overloaded semantically to represent logical relation, negation, and truth value. Disambiguating the meaning of an expression is placed at the interpretative interface; the syntactic transformation mechanism needs to address only one token.

The resulting system has the advantage of rapid calculation using an abstract representation which eliminates the redundancies inherent in PC.

In order to express the mathematics of LoF in a programming language, we have mapped the primitive LoF token of distinction onto the concept of a list in LISP. Losp is a computational language with these properties:

1. Losp expressions are fully compatible with LISP expressions, but have a different interpretation. Losp and LISP codes can be freely intermixed.

2. The joint Losp/LISP system combines the computational convenience of LISP functions with the computational rapidity of Losp abstraction.

3. Losp transformation and simplification rules can be interpreted as machine-level instructions in the LISP environment.

The following Section on *The LoF Formalism* develops the ideas inherent in mathematical symbolization from first principles. The discussion begins by identifying proto-symbolic concepts underlying the act of representation with symbols. The approach is entirely constructive. From these primitive ideas, we reason from intuition to develop the rules (axioms) that govern symbolization. (In his book, Spencer-Brown presents the mathematical treatment of these axioms.) In the next Section, *Morphism with Boolean Algebra*, we expand these concepts to include variables and show that such a system is morphic with Boolean algebra. We then examine some limited aspects of Predicate Calculus as an interpretation of LoF. The organizing principle for this interpretation is to demonstrate that the complexities of transformations and methods of proof in Predicate Calculus can be seen to reduce to simple structures in the lexicon of LoF.

This monograph is incomplete; a thorough discussion of Losp algorithms, of functional and relational forms in Losp, and of applications has been addressed in separate papers.

THE LOF FORMALISM

The Laws of Form [G. Spencer-Brown, 1972] specifies the axioms and theorems for both the arithmetic (without variables) and the algebra of LoF. This work has had a controversial history, as would be expected by a system that is claimed to conceptually simplify the traditional field of logic. The controversy centers around whether or not a new representation per se can contribute greater understanding to a mathematical formalism. There is no doubt that the primary arithmetic (as Spencer-Brown calls his basic system) is

> "... up to isomorphism, given by join and addition in the two-element Boolean algebra"
>
> [Banaschewski, 1977]

Traditional reviewers claim that the Laws of Form

> "... appears to be simply another axiomatization of Boolean algebra."
>
> [Gould, 1977]

However,

> "A change of context and style (notation) may reveal the intuitive and conceptual underpinnings of a field, in a way that other, formally equivalent systems, do not."
>
> [Kauffman and Varela, 1980]

Others view Spencer-Brown's work as

> "... not an arbitrary new calculus, but that particular calculus which can let us see deeper into the nature of mathematics"
>
> [L. L. Whyte, 1972].

We choose not to enter the controversy at the level of foundations of mathematics, rather we wish to evaluate Spencer-Brown's work from a purely practical perspective. If this controversial logic is indeed revolutionary, then it should yield specific advantages when applied to fields of research relying on formal logic. Automated reasoning is such a field: it uses Predicate Calculus as a primary form of representation, inference techniques as a primary form of reaching answers, and is currently experiencing the need for faster and more efficient algorithms. As well, significant gains in automated reasoning will have significant impact on several fields within computer science, such as expert systems, theorem proving, logic circuit design, program verification, and real-time systems control.

THE LAWS OF FORM

Bertrand Russell said that Spencer-Brown

"... has revealed a new calculus of great power and simplicity"
[L. L. Whyte, 1972]

Because the basic concepts of the Laws of Form are fundamentally new, it is often difficult to convey their full generality and power. Our approach is to describe these concepts and to illustrate their utility with examples.

In order to concentrate on description, we omit all proofs in the pages that follow. It is important to realize, however, that the Laws of Form, and Losp without autonomous states, are semantically complete axiomatizations of Boolean Algebra. In the Laws of Form, Spencer-Brown proves the completeness and consistency of the formalism, and shows that the primary axioms are independent. Elsewhere [Schwartz, 1981], this formalism is shown to be isomorphic with Propositional Calculus.

Although the following discussion may appear to be unrelated to logical formalism, it is actually the source of Spencer-Brown's ability to simplify logical representations. The remarkable property of these few elementary distinctions is that they are precursors to logic, and thus are the basis from which Propositional Calculus evolves. Please note that almost everything to be presented is at the level of obvious mathematical knowledge. Most of these ideas are buried in the conventions of symbol manipulation. The strength of the current approach is to make these conventions explicit, thus alerting the user of the symbol system to the details of simple symbolic acts. As it turns out, knowledge of these details permits both clarification and simplification of symbol manipulation.

Proto-Symbolic Concepts

"We begin at a point of such degeneracy as to find that the ideas of description, indication, name, and instruction can amount to the same thing."
[Spencer-Brown, 1972]

By beginning at the point where description and injunction meet, Spencer-Brown proposes a formalism that is blind to the procedural-declarative controversy.

The key to understanding the three fundamental concepts of the Laws of Form is to imagine the simplest possible representational system. At first such a system contains only a sentient "mathematician", the symbol manipulator. The initial act of formal representation requires that the mathematician set aside a space which will contain all symbolic representations. This space is totally without content. The three proto-symbolic concepts are then:

1. THE INITIAL DISTINCTION: The setting aside of a space that will contain all symbolic representations.

2. THE FORMAL SPACE: the space set aside to contain the symbolic representations.

3. THE INTERPRETIVE CONTEXT: the space of the mathematician, from which the formal space has been distinguished.

The formal space is the domain of syntactic objects; the interpretive context is the domain of semantic content; the initial distinction is the relation that unites syntax with semantics.

In order to treat these concepts mathematically or computationally we must agree upon symbols to represent them. At this point, Spencer-Brown makes his first innovation. Both the formal space and the interpretive context are represented by the empty space, a non-symbol. It is the responsibility of the mathematician to know which is which. Practically, this is not a difficulty. When we engage in symbol manipulation, for example on a sheet of paper, we conventionally know that the unmarked page is the formal space, the boundary between the page and the rest of the world (i.e. the edges of the paper) is the initial distinction, and the (unmarked) air surrounding the page is the interpretative context which contains the mathematician and the meanings that the mathematician intends for the symbols drawn on the page.

Spencer-Brown's second innovation is to introduce self-reference at the heart of his representation theory. The formal space can contain a representation of (i.e. a token for) the initial distinction. In this way, a formal space can be used to symbolically examine the initial distinction and the contained formal space itself.

The Mark of Distinction

The token that represents the initial distinction in the formal space is called a MARK. This proto-symbol is the only explicit symbol used in the arithmetic (that is, the formal system without variables) of LoF.

Being the only type of token, the mark is totally generic. Its particular shape is arbitrary. The function of a mark is to make a DISTINCTION. At this point, before semantics is introduced, the only distinction is between the context of interpretation and the space of symbols.

Placing a mark in the formal space (that is, drawing a mark on an empty sheet of paper) introduces several subtle conventions that, in effect, define the act of symbolizing:

1. The initial distinction remains, of course, as the edges of the formal space (the boundary of the paper).

2. The token for the initial distinction is a symbolic representation of the initial distinction. Its meaning is quite abstract: it symbolizes the act of symbolizing. The initial act in the process of symbol manipulation is self-referential.

3. The mark must be taken as the NAME of the formal space. Since it is the only landmark within the formal space, the mark is the only means of identifying the formal space.

4. Drawing the mark introduces the distinction between a thing and its label. This is similar to considering the token for the word "penguin", rather than our usual convention of considering the meaning of the word penguin. A penguin has feathers while a "penguin" has seven letters. The mark distinguishes the empty page, while the "mark" on a page cancels the emptiness.

5. Thus, the act of marking introduces the fundamental dichotomy between emptiness and existence.

6. For the purposes of communication, the mark is an instruction to duplicate the intentions of its author. Since the only intention at this point is to initialize a symbolic space, the functional intent of the mark is to request an act of imagination: let's represent our context here. In other words, let's cancel the void and create a world.

By focusing upon distinction, Spencer-Brown places formal emphasis on DIFFERENCE, achieving a mathematics that embodies the philosophies of William James ("There is no difference except the difference between difference and no difference") and Gregory Bateson ("Only difference makes a difference").

Parens

We will use the parenthesis as a token of the initial distinction, calling it PARENS. The inside of the parens is the formal space; the outside is the interpretative context. Thus:

```
                       distinction
                       |         |
                       v         v
                       (         )
            ^                ^                  ^
            |                |                  |
interpretative context   formal space     interpretative context
```

Several novel properties are incorporated in this primitive representation of the process of symbolization:

1. The distinction is the boundary of the formal space. A linear, typographical representation requires this boundary to be bilocated, thus the apparent two sides of the parens. View the parens, however, as a single token.

2. Similarly, the separation of the interpretive context is an artifact of the linear representation. There is one context, and it extends off the edges of the paper and into our third dimension.

3. The interpretative context PERVADES the formal space. Like the distinction, it is of a higher dimension than the formal space and has access to all parts of it, just as a mathematician has access to all his symbols.

4. The formal space is literally empty, it is not a Cartesian space full of points.

We will also refer to a concept as IRRELEVANT, when, at the current stage of development, the mathematically familiar concept does not yet exist. Almost all common concepts, at this point, in fact are irrelevant. The parens signifies a Universe empty of both objects and concepts. In the following Sections, as they are needed, we explicitly introduce all the concepts in the mathematics of LoF.

The Empty Formal Space

To avoid confusion when referring to the empty formal space, we will introduce a new kind of token to draw our attention explicitly to the characteristic of emptiness. Typographically, the token is ><. This token cannot be named explicitly since it is technically non-existent. For convenience we will use a meta-symbol, <void>. <Void> is forbidden to interact in any way with any token, including itself.

The rules for representing <void> are innovative. It is, technically, a totally irrelevant symbol within the context of the formal space. We use it solely to remind the reader (or the mathematician) that the empty space exists. It is an empty, virtual token.

It is important to keep in mind that emptiness is present in all unmarked parts of the formal space; <void> can be added arbitrarily anywhere within the formal space at any time. The token for <void>, however, has no syntactic meaning. Its sole function is to point out the obvious, that there is a sheet of paper underneath the symbols.

Atomic States

The parens token, ( ) , and the empty token, >< , form the atomic, or elemental, states of the Laws of Form. They form a bottom for all calculations.

Arrangements

We have introduced one token, parens, which constitutes the entire lexicon of the arithmetic of LoF. Since <void> is a non-interactive token, all expressions are built up from parens alone. There are two locations which support the construction of additional parens, inside the initial parens and outside the initial parens. Thus, there are two distinct arrangements composed of exactly two parens:

```
( ) ( )                 construction on the outside

( ( ) )                 construction on the inside
```

Both constructions introduce an extension of the concept of the initial distinction. Construction on the outside can be understood as the introduction of a new formal space, totally independent of the initial formal space, much like picking up a fresh, clean sheet of paper. Construction on the inside establishes the form for reflection; the token for a distinction is recursively entered into a token for the initial distinction.

Construction on the outside is the means by which the LoF symbolism introduces new objects. Construction on the inside is the means by which complexity is introduced into existing objects. It embodies the notions of hierarchical containment and set membership.

As well, construction on the inside explicitly introduces the concept of RELATIVE CONTEXT. In an expression such as

```
( >< ( ) )
```

the inner parens labels a formal space within the relative context of the outer parens. We draw attention to this relative context by the <void> token. The outside of both is, then, the absolute context of the symbol creating mathematician.

As will be seen in the Section entitled *Objective and Functional Perspectives*, construction on the outside is archetypical of an world of objects that are countable. Construction on the inside is archetypical of a world of functions that are doable. The former foreshadows the symbol manipulation of mathematicians; the latter foreshadows the functional evaluation of computer scientists.

We can continue the process of construction, providing an unlimited number of distinct arrangements. Some examples:

```
( ) ( ) ( ) ( ) ( )

( ( ( ( ( ) ) ) ) )

( ( ) ( ) ) ( ( ) )
```

The construction of arrangements thus far has been entirely syntactic. We will call an arrangement an EXPRESSION when it has an attached semantic meaning.

Well-Formed LoF Arrangements

The Backus Normal Form for well-formed LoF arithmetic arrangement is:

```
        ><          ::=         |    ><

     <paren>        ::=    ><  (  ><  )  ><

  <arrangement>     ::=    ><  |  <paren>

                               |  <arrangement> <arrangement>

                               |  ><  (  <arrangement>  )  ><
```

Alternatively, the construction of well-formed arrangements can be prescribed recursively:

Basis: >< is a well-formed parens arrangement

Generating rule: if A and B are well-formed,

then both (A) and A B are well-formed.

Closure: There are no other well-formed parens arrangements.

Space

The formal space is blind to the location of tokens within it. Although a linear typographical presentation forces an ordering upon the arrangements within a space, this ordering is an unnecessary constraint on the LoF form, and until introduced as a restriction, ordering is irrelevant. We use the term SHARING to refer to the non-ordered relation between arrangements in space. Thus, for example,

( ) (( )) is identical to (( )) ( )

That is, order in LoF is irrelevant.

Interpretive Intent

Whenever a mathematician (or anyone else) enters a symbol into a formal space, this entry is made with an intention to assign a value or a meaning to the symbol. We assume fundamentally that every mark has an intended meaning which can be accessed by interpreting the symbol. With this in mind, the following transformation rules can be understood as canonical limitations on the syntax of all symbol systems. Without them, the consistency of interpretative intent is lost.

TRANSFORMATION RULES IN THE ARITHMETIC OF LoF

The informal construction techniques discussed thus far are sufficient to determine a set of transformation rules, or axioms, for well-formed parens arrangements. These rules are syntactic, and are independent of any semantic interpretation placed upon the symbology. However, in order to provide a concrete feeling for the necessity of these transformation rules, we will continue to refer to the interpretation of the mathematician constructing symbols on a blank page.

Meta-Relations

A meta-relation is a relation between formal spaces. In a sense, the meta-symbol that represents a meta-relation exists in the interpretative context, outside of the formal space. Meta-relations represent actions on the part of the mathematician on the entire formal space, converting it into a different configuration of symbols. The meta-relations introduced here are EQUALS and STEP.

We use the traditional meta-symbol = , EQUALS, to indicate permissible transformations between arrangements. Equals is bidirectional; it grants permission to convert the arrangement found on one side to the arrangement found on the other side. We can say that the equals sign joins two formal spaces, each of which holds equivalent arrangements. We will temporarily adopt the convention that the motive of the mathematician is to simplify arrangements, and will put the simpler arrangement on the right-hand side of the equals sign. Clearly, though, an equals meta-relation also allows the construction of complex arrangements from simple ones.

We use the meta-symbols ==> and <== to indicate directional steps taken in a transformation permitted by an equals meta-relation. A STEP may be viewed as an object substitution (from a declarative perspective) or as a computational process (from a procedural perspective).

Whenever a meta-symbol is used, we invoke a convention of linear representation: we permit two formal spaces, and their contents, to coexist on the same typographical line.

Identity and Non-Permissible Transformations

In order to assure the consistency of interpretive intent, we invoke the concept of IDENTITY:

```
                  ( )   =   ( )
and
                  ><    =    ><
```

Since parens and <void> are the only primitive states, we have a clear definition of non-permissible atomic transformations. There is only one:

```
                  ( )   ≠   ><
```

where the meta-symbol ≠ means DOES-NOT-EQUAL, or, operationally, if you make this transformation, then you have undermined the consistency of the representation.

Consistency and Uniqueness

The transformation rules that follow define the permissible transformations in the arithmetic of LoF. That is, whenever they are applied, the non-permissible transformation, Does-not-equal, does not occur. These rules are a CONSISTENT axiom set.

When successively applied to an arrangement, these axioms eventually transform the arrangement into an atomic state that is UNIQUE to that arrangement. The unique atomic state associated with each particular is called the VALUE of the arrangement.

The Two Transformation Rules

Replication and *Involution* are the independent axioms of LoF. We refer to them as transformation rules in order to focus on the utility of the axioms for simplification of expressions. All arrangements can be reduced to one of two atomic states (parens or <void>) by the application of these two rules.

For each transformation, three names are necessary: one will name the transformation as a unified concept, one will name the step that transforms an expression from complex to simple, and one will name the step that transforms an expression from simple to complex. The following two transformations are all that are needed to totally specify all operations in the arithmetic.

Replication

( ) ( )	=	( )	REPLICATION
( ) ( )	==>	( )	CONDENSE
( ) ( )	<==	( )	CONSTRUCT

The mathematician is free to add new pages without changing the intended meaning of the previous pages. Spencer-Brown's name for this rule is CALLING.

A symbolic way of understanding this rule is to notice that the two parens on the right-hand side are indistinguishable. Whatever they mean, they mean the same thing, since we cannot tell them apart. This rule is the formalization of the common convention that specific symbols do not change their meaning from one step to the next. All parens are identical; duplicity of representation communicates nothing new.

CONDENSATION specifically undermines the concept of number. CONSTRUCTION is the prototypical successor operation.

Another way of viewing this rule is to impute some unlabelled operation inherent in the formal space that maps arrangements onto arrangements. Naming this imputed function is one way of placing an interpretation on the symbol structures being developed. The unique characteristic of this notation is that the functional impact of the formal space is represented not by an infixed token but by the coexistence of expressions in the formal space itself. Coexistence in a formal space, SHARING that space, abstracts away concepts of commutativity and associativity, therefore its functionality is constrained to operations on commutative (Abelian) semigroups. Replication, in this context, specifies the IDEMPOTENCY of the formal space.

We can check the consistency of the rule of Replication with <void>:

```
                  ><   ><   =   ><
```

which is valid by definition.

Involution

```
                  ( ( ) )    =    ><           INVOLUTION

                  ( ( ) )   ==>   ><           COLLAPSE

                  ( ( ) )   <==   ><           CREATE
```

By itself, <void> represents the absolute interpretive context (it says that the formal space is not only empty, it is as yet unrecognized). The rule of Involution can be interpreted to say that the token for a relative distinction (the inner parens) has no properties when viewed from the interpretative context of an absolute distinction (the outer parens). The mathematician who examines the meaning of the empty page will find nothing. Spencer-Brown's name for this rule is CROSSING.

COLLAPSE specifically undermines the concept of recursion. CREATE is the prototypical operation of self-reference.

Another viewpoint: A distinction made again is not the same as the distinction, therefore construction on the inside cannot be equal to the original construction (the single parens). The only alternative is for it to be equal to <void>.

The rule of Involution applies to <void>, again by definition:

```
                  >><<   =   ><
```

Replication is easily understood from the declarative perspective; Involution is most easily understood from the procedural perspective, which will be introduced in the following sections.

Objective and Functional Perspectives

The preceding discussion has implicitly taken an objective perspective. <void> and parens were treated as objects that were either constructed or replaced (via substitution) with equivalent objects. In fact, the technique of substitution of equals is unique to an object-oriented perspective. Alternatively, a parens expression can be viewed as a functional set of instructions to be enacted by the interpretive context. Functionally, parens is *Bounding*.

> "If a state can be indicated by using a token as a name, it can be indicated by using the token as an instruction subject to convention. Any token may be taken, therefore, to be an instruction for the operation of an intention."
>
> [Spencer-Brown, 1972]

Erasure: Functional Interpretation of the Formal Space

Each transformation rule can be achieved without the use of equals by the act of erasure. Thus, for Replication, either of the existing parens can be erased to achieve the same result as transformation by substitution. For Involution, both parens should be erased. This technique provides a unique simplification methodology, by destructive alteration of an expression rather than by substitution of equivalent expressions.

The operation of erasure treats all parens as objects. It could be viewed as the substitution of non-existent objects (<void>) for existent objects (parens). This is not particularly satisfying since it forces the introduction of the concept of an imaginary object. A more appealing view is to associate an implicit function with the formal space itself. All arrangements in the formal space are then the arguments of the formal space. In other words, the formal space is assigned a functional interpretation, called SHARING, with the following mappings:

```
( )          ==>   ( )

( )  ( )     ==>   ( )

( ( ) )      ==>   ><
```

However, another form of functional evaluation is available, in which the parens rather than the formal space is assigned a functional interpretation.

The Functional Interpretation of Parens

Rather than treating a parens as an object within an interpretative context, it can be treated as a function that acts on the contents of the formal space as arguments. This function is "change the state of the contents". There are only two atomic states, parens and <void>; the parens function toggles the representation between the two. The mapping is:

```
(   ><   )   ==>  ( )

(  (  )  )   ==>   ><
```

In the first line, the function parens acts upon its contents (<void>) to yield the state parens. In the second line, the function parens acts upon its contents (parens) to yield the state <void>. When a parens is added functionally, it is a bounding function.

Bounding is the prototype for the concept of COMPLEMENTARITY.

An alternative way to read the arrangement

```
( ( ) )
```

is to see it as the composition of two bounding functions. In this light, the rule of Involution looks like double negation.

An innovative idea has been introduced here: a single token can have two semantics, as a state or as a function. These semantics can be freely intermixed, and can even be assigned to tokens arbitrarily, without loss of representational coherence. In other words, the interpretations are independent. Both parens and <void> can be either functions or states. The important idea is that the assignment is at the discretion of the mathematician in the interpretive context; it is irrelevant to the outcome of the syntactic computation. Tokens are both descriptions to be noted and instructions to be followed.

Involution as the Initial Distinction

From the declarative perspective, the token for the initial distinction is the parens. This distinction is symbolized by marking a single parens on the empty page. Alternatively, from the functional perspective, the token for the initial distinction is (( )) , the double parens of Involution. Rather than the initial act of symbolization breaking the emptiness of the virgin page, it could equally as well be seen as the functional elaboration of nothingness.

The difference between these approaches is the explicit or implicit recognition of the edges of the paper. From the declarative perspective, the edge of the page represents itself; it is symbolically implicit. The initial symmetry breaking act of symbolization, marking a parens, initializes the formal space as the rest of the paper. From the functional perspective, however, the edge of the interpretive context is explicitly represented by the outer parens of Involution. All further symbolic activity takes place within this outer mark.

Generalization Over Steps

Replication can be generalized to any number of parens:

```
( )  ( )  ( )  ...   =   ( )
```

This can easily be seen as the successive application of the simple rule of Replication, exactly N - 1 times, where N is the length of the arrangement of parens on the right-hand side.

Involution can be generalized to any even depth of nesting of parens:

```
( (  ( (  ( (  ...  ) )  ) )  ) )   =   ><
```

This can be seen as the successive application of the simple rule of Involution, exactly N/2 times, where N is the depth of nesting of the arrangement on the right-hand side.

EXAMPLES OF SYNTACTIC SIMPLIFICATION

From a declarative perspective, simplification takes place by the substitution of simple expressions for complex expressions. The transformation rules of Replication and Involution specify all acceptable substitutions. From a functional perspective, simplification takes place by the evaluation of nested arrangements of parens as bounding functions acting on contents. Simple values evolve from complex functional arrangements. Since a perspective is a matter of choice on the part of the interpreter, simplification by substitution or by evaluation is also a choice; these actions are essentially equivalent.

To demonstrate the two perspectives on simplification, consider the arrangement:

```
(  ( )  ( )  ( ( ) )  )
```

Simplification   by Substitution

Whenever a sub-arrangement matches the configuration on the right-hand side of a transformation rule, that rule can be applied to simplify the total arrangement. Thus, by substitution,

```
            (  ( )  ( )  ( ( ) )  )
      ==>   (  ( )       ( ( ) )  )       Replication
      ==>   (  ( )          ><    )       Involution
      ==>      ><                         Involution
```

Note that the two rules are independent, thus the order of application is irrelevant. As well, the contents of any formal space (here we focus on the space nested at a depth of 1) are unordered, so their typographical ordering is irrelevant.

Simplification   by Functional   Evaluation

Each parens is interpreted as a bounding function on its contents. Evaluation must begin in the deepest space of the expression. For clarity, the depth of each typographical zone is indicated by a number:

```
            (   (   )   (   )   ( (   ) )   )
          0   1   2   1   2   1  2  3  2  1   0
```

The state of the deepest space is, naturally, empty. The technique is to successively ascend, in parallel, from the deepest space to the shallowest space (the interpretative context), while changing the state of the solution at each change of depth. To illustrate this process, let N stand for the Non-marked state (<void>), and let M stand for the Marked state (parens). The evolution is:

```
            (  ( )  ( )  ( (N) )  )
      ==>   (  (N)  (N)  (  M  )  )       depth = 2
      ==>   (   M    M      N     )       depth = 1
      ==>                N                depth = 0
```

The number of steps is equal to the greatest depth of the arrangement being simplified. The functional mappings are the same as in the Section on *The Functional Interpretation of Parens*, specifically:

```
            (N)   ==>   M

            (M)   ==>   N
```

In stepping from depth = 1 to depth = 0, the rule of Replication is used in its functional interpretation. The arguments of any specific parens are a combination of Ns and Ms. In the case of a mixture of both Ns and Ms (as in the present example), the Ns, being non-existent place-holders, are irrelevant and can be omitted. In the case of multiple Ms, the parens can act on any one of them since they are all identical. The same idea applies in the case of multiple Ns.

We have introduced another innovative characteristic of this formalism: the arity of the parens bounding function (or the implicit SHARING function assigned to the formal space) is irrelevant. This simplification of the conventions of functions is available because any combination of arguments can be transformed into a single explicit argument. There may be, of course, any number of non-marked arguments corresponding to the occurrences of N in the expression, but their numerosity is irrelevant precisely because these arguments are treated by the formalism as non-existent. In light of this relaxation, we will refer to the CONTENTS of a parens (or of a formal space) rather than to the arguments of a function.

GENERALIZATION TO VARIABLES

The introduction of variables converts a formalism from an arithmetic to an algebra. Variables represent an arbitrary arrangement within the context of other arrangements. We will use the set of lower-case letters {a, b, c, ...} to represent abstract forms that are *either one* of the ground states. It is convenient to think of a lower case letter as a placeholder for either <void> or ( ).

If a particular ground state in a larger arrangement does not impact the unique atomic state of the larger arrangement, then a variable may be substituted for the ground state in an act of generalization. Thus, for example, we have seen that both of these equations hold:

```
                  ( )  ( )  =  ( )
and
                   ><  ><   =  ><
```

These two equations can be expressed succinctly by the generalized form of Replication:

```
                    a  a    =   a              REPLICATION  (algebraic)
```

Inversely, the above equation with variables can be tested for consistency by the casewise substitution of parens and <void> for the variable token *a*. We will illustrate this with the generalized version of Involution:

((*a*)) = *a*   **INVOLUTION (algebraic)**

Case I: let *a* = ><

Substituting: ((><)) = ><

which is the arithmetic version of Involution.

Case II: let *a* = ( )

Substituting: ((( ))) = ( )

which is consistent since

((( ))) ==> (><).

There are no other cases, so the generalized version of Involution is consistent.

Dominion

Another important equation to be derived from Replication is this special case:

( ) ( ) = ( )

and

( ) >< = ( )

Therefore: ( ) *a* = ( )   **DOMINION**

This theorem suggests that variables do not exist in the interpretative context of an empty formal space. That is, the atomic parens dominates all variable arrangements, and therefore all arrangements.

Dominion is the form by which tokens are introduced into a formal space. This can be followed most clearly from the functional perspective: both steps of this proto-evolution of symbols represent the activity on the part of a mathematician to create a symbolic world:

```
>< ==> ( ( ) )

   ==> ( a ( ) )
```

Pervasion and Position

Another single variable theorem is of fundamental importance:

```
( a )  a   =  ( )  a          PERVASION
```

A defining characteristic of the interpretative context is that it is pervasive, it exists both outside and inside the parens distinction. This PERVASION extends to the contents of the interpretative context: if an arrangement exists in the context, then it can also exist in the formal space.

The schema for the act of symbolizing is extended thus:

```
><  ==>  ( ( ) )  ==>  ( a ( ) )  ==>  ( a (a) )
```

First the formal space is distinguished; then an arrangement is conceived in the interpretative context and is transferred symbolically into the formal space. It will be convenient to label this theorem:

```
( a (a) )  =  ><              POSITION
```

To elaborate the empty double parens of Involution by the creation of a variable is to establish a position with respect to that variable. The token a, in this case, identifies the chosen position.

Reflexivity

We will abbreviate the transformation sequence of Pervasion and Dominion,

```
(a) a  ==>  ( ) a  ==>  ( )
```

by the theorem of REFLEXIVITY of distinction:

```
(a) a  =  ( )                 REFLEXIVITY
```

Note that POSITION is the dual of REFLEXIVITY.

Distribution

The following rule of DISTRIBUTION is the only axiom not yet presented:

```
a ((b) (c))  =  ((a b) (a c))       DISTRIBUTION
```

This transformation rearranges the variables in an expression without simplifying the expression. The rearrangement is from two expressions sharing the same space to one expression in that space. Using the axiom of Distribution, it is possible to rearrange any complex expression into a form contained by a single parens token. Thus, Distribution permits transformation between the two types of initialization of a symbolic space.

Distribution is the only axiom containing multiple variable tokens.

The Bridge

We introduce here a theorem that connects the arithmetic of form to the algebra:

> Two arrangements are equal
> if they are equal for all possible states of one variable.

There are two ways to evaluate an equation: one is by algebraic manipulation, the other is by examination of the equation by cases, one case for each possible value of any chosen variable. It is often not necessary to consider more than one particular variable, if that variable is judiciously chosen.

Proof of Distribution by Appealing to the Arithmetic

We will illustrate the use of the BRIDGE in proving that the rule of Distribution is a tautology.

Case I: let a = ( )

```
substitute:       ( ) ((b)(c))  =  ( (( ) b) (( ) c) )
```

By Dominion and Involution, these both reduce to ( ).

Case II: let a = ><

```
substitute:       ><  ((b)(c))  =  ( (><  b) (><  c) )
```

both of which are identical to ((b)(c)).

Therefore the rule of Distribution is confirmed.

We have included this confirmation by evaluation of both possible states of a variable to illustrate one technique for simplifying LoF expressions. The technique is an appeal to the arithmetic.

Other Theorems

The following theorems can be derived by appealing to the arithmetic, or they can be derived by direct algebraic manipulation from the axioms.

((a) b) a = a	OCCLUSION
(((a) b) c) = (a c) ((b) c)	DEPTH
(((a)(b)) (a c)) = ((a) b) (a (c))	FLEX

These theorems are useful for rearranging complex algebraic expressions. Naturally many other theorems could have been listed.

Minimal Arrangements

Two theorems about algebraic expressions are particularly interesting:

1. Any given expression can be transformed into an equivalent expression not more than two parens deep.

2. Any given expression can be transformed into an equivalent expression with not more than two occurrences of a particular variable.

These theorems specify bounds on the manipulation of expressions. The two extreme cases can be considered dual canonical forms for LoF expressions.

GENERALIZATION TO ARBITRARY ARRANGEMENTS

We introduce capital letters {A, B, C,...} to represent arbitrary arrangements of parens and variables. While small letter variables stand in place of either ground value, capital letter variables stand in place of any arrangement. The axioms and theorems of LoF can all be directly generalized to capital letter variables by substituting capital letters for small letters in a valid equation.

The generalization is possible because any LoF arrangement will reduce to a ground state when all small letter variables are bound to particular values. Equations that are valid for small letter variables are still valid for arbitrary arrangements, since all arrangements will reduce to a small letter variable. Thus

$$a \ a = a$$

is the Replication Rule for ground states. Replication also applies when any arrangement is replicated. The small letter variables in the replicated arrangements will each have exactly the same behavior in each replication. Thus:

```
A  A  =  A
```

For example, let A = (a (b))

```
(a (b))  (a (b))  =  (a (b))
```

This equation will be true for any consistent binding of variables *a* and *b*.

A Summary of the Properties of the Laws of Form

The axioms and theorems of LoF can now be stated in their generalized form, as applying to all arrangements that match the particular pattern of the rule. This turns LoF into a generic pattern-matching language.

===

REPLICATION	A A	=	A
INVOLUTION	((A))	=	A
DOMINION	( ) A	=	( )
PERVASION	(A B) A	=	(B) A
DISTRIBUTION	A ((B)(C))	=	((A B) (A C))
OCCLUSION	((A) B) A	=	A
DEPTH	(((A) B) C)	=	(A C) ((B) C)
FLEX	(((A)(B)) (A C))	=	((A) B) (A (C))

===

FORM EQUIVALENCIES

Technically, three of these equations (Involution, Dominion and Pervasion) are sufficient for a complete axiom set. The rest are derivable from them.

Summary of Notational Characteristics

1. The single token parens, ( ) , and the empty token <void>, >< , are overloaded with meanings as both functions and states.

2. The empty token carries an implicit interpretation that is dual to the explicit interpretation of parens.

3. Interpretation of tokens from either a functional or an declarative perspective is arbitrary at all times.

4. Arrangements are abstract; properties of associativity, commutativity, and arity are implicit and irrelevant.

5. The system is equational.

MORPHISM WITH PROPOSITIONAL CALCULUS

The Laws of Form have been shown to be morphic with Boolean Algebra [Banaschewski, 1977]. From this result, several applications, or interpretations of the LoF arrangements and axioms become available. We will list three of these, and then concentrate specifically on the morphism between LoF and Propositional Calculus.

===

LoF	*Boolean Algebra*	*Algebra of Subsets*	*Switching Circuits*	*Propositional Logic*
States				
><	lower bound	null set	closed	contradiction
( )	upper bound	universal set	open	tautology
Functions				
><	join	union	parallel	disjunction
( )	inversion	complement	opposite	not
(( )( ))	meet	intersection	series	conjunction
Relations				
( )	less than or equal to	subset of	implies	implies

===

MAPPINGS

Note that LoF forms can have multiple usage, as functions, as states, or as relations. The last row is the partial order relation for the various applications.

In this Section we will first establish a mapping between LoF token arrangements and the traditional lexicon of Propositional Calculus (PC). We will illustrate how the various techniques for simplification of Losp expressions can be applied to problems in PC and how these techniques deeply simplify the evaluation strategies in PC, such as truth tables. We will then illustrate how proof techniques in PC are also simplified.

MAPPING OF STATES AND FUNCTIONS

There are two primitive states in LoF, parens and <void>; similarity there are two truth values in PC, TRUE and FALSE. The choice of association between these sets is arbitrary, and will vary depending upon the specific application. Once the association is chosen, however, all other mappings follow deterministically.

We will use only one mapping between states. Associate

```
                  ><    with    FALSE,
and
                  ( )   with    TRUE.
```

The functions of PC then map directly onto arrangements of parens interpreted functionally.

NOT

When parens is read as a function, and when the state mappings given above are incorporated, the equation

```
                     (  ><  )     =   ( )
becomes
                  BOUNDING FALSE  =  TRUE.
```

It can easily be seen that the interpretation of the BOUNDING function maps onto the PC function NOT. For verification, note that

```
                  (  ( )  )   =   ><
```

That is, BOUNDING TRUE = FALSE.

OR

Read

```
                  ( )  ><   =   ( )
as
                  SHARING TRUE, FALSE  =  TRUE.
```

The function SPACE (or <void>) is the PC function OR. Since the arity of SHARING is arbitrary, it corresponds to an n-ary OR.

AND

Read

```
                  ( ( >< ) ( >< ) )   =   ><
as
            NOT [ NOT [FALSE] OR NOT [FALSE] ]  =  FALSE.
```

By DeMorgan's Transformation, the PC expression on the right is equivalent to FALSE *AND* FALSE.

IMPLIES and EQUIVALENT in PC are derivable in a similar intuitive way. In summary, and using variables, the mapping are:

==

PC NAME	*PC TOKENS*	*LoF TOKENS*
TRUE	t	( )
FALSE	f	><
NOT	not[a]	(a)
OR	or[a, b]	a b
AND	and[a, b]	((a)(b))
IMPLIES	implies[a, b]	(a) b
EQUIVALENT	equiv[a, b] or a = b	(((a) b) ((b) a)) or (a b) ((a)(b))

==

TABLE OF CORRESPONDENCE

Logical equivalence has two possible parens forms that are interchangeable by using the FLEX transformation rule.

LOGICAL TRANSFORMATIONS

One method of demonstrating the simplifying power of LoF is to show that many of the transformation rules necessary within PC are condensed into irrelevancy within LoF. The PC transformations that are relevant to LoF follow:

==

Name *Functional Notation*	*PC Expression*	*LoF Arrangement*
Double negation		
not[not[a]] = a	¬(¬a) = a	((a)) = a
Duplication		
or[a,a] = a	a V a = a	a a = a
and[a,a] = a	a & a = a	((a)(a)) = a
Excluded Middle		
or[a, ¬a] = true	a V ¬a = t	a (a) = ( )
Distribution		
or[a, and[b,c]] = and[or[a,b], or[a,c]]	a V (b & c) = (a V b) & (a V c)	a ((b)(c)) = ((a b)(a c))
and[a, or[b,c]] = or[and[a,b], and[a,c]]	a & (b V c) = (a & b) V (a & c)	((a)(b c)) = ((a)(b))((a)(c))
Biconditional Exchange		
equiv[a,b] = and[implies[a,b], implies[b,a]]	(a == b) = (a -> b) & (b -> a)	(((a) b)((b) a)) = ( )

==

RELEVANT (STRUCTURAL) TRANSFORMATIONS

As is illustrated by the above table, the fundamental transformations of LoF map onto logical transformations as follows:

Involution	<==>	Double Negation
Replication	<==>	Duplication
Distribution	<==>	Distribution
Pervasion and Dominion	<==>	Excluded Middle

Note that the dual forms of Duplication and Distribution are unnecessary in LoF. Within LoF, these duals are merely the same operation with different objects (and additional invocations of Involution).

The function of Biconditional Exchange in the equational syntax of LoF is to shift all variables to one side of the equation.

Irrelevant Transformations

The transformations in PC that are irrelevant to LoF fall into three categories (next page):

1. Transformations subsumed by the structure of the formal space:
   Commutation and Association.

2. Transformations subsumed by Involution:
   DeMorgan, Contraposition, and Exportation.

3. Transformations inherent in the LoF representation:
   Conditional Exchange.

A large part of the transformational machinery of PC can be seen to be irrelevant and unused in the LoF representation. Thus, a large savings in number of operations is realized by use of the LoF representation. To illustrate this, we next demonstrate how the clumsy and resource consumptive truth table method of PC can be entirely eliminated in favor of a direct algebraic determination of truth values in LoF.

```
==========================================================================

Name
  Functional   Notation      PC  Expression      LoF  Arrangement

Commutation

  or[a,b] = or[b,a]           a V b = b V a       a  b , order irrelevant

  and[a,b] = and[b,a]         a & b = b & a     ((a)(b)) , order irrelevant

Association

  or[a, or[b,c]] = or[or[a,b], c]

            a V (b V c) = (a V b) V c           a b c , grouping irrelevant

  and[a, and[b,c]] = and[and[a,b], c]

            a & (b & c) = (a & b) & c           ((a)(b)(c))

DeMorgan

  not[or[a,b]] = and[not[a], not[b]]

            ¬(a V b) = ¬a & ¬b                  (a b) = (((a))((b)))

  not[and[a,b]] = or[not[a],not[b]]

            ¬(a & b) = ¬a V ¬b                  (((a)(b))) = (a)(b)

Contraposition

  implies[a,b] = implies[not[b], not[a]]

            (a -> b) = ¬b -> ¬a                 (a) b = ((b)) (a)

Conditional  exchange

  implies[a,b] = or[not[a], b]

            (a -> b) = ¬a V b                     (a) b = (a) b

Exportation

  implies[and[a,b], c] = implies[a, implies[b,c]]

            ((a & b) -> c) = (a -> (b -> c))    (((a)(b))) c = (a)(b) c

==========================================================================
                   TRIVIAL  (STRUCTURAL)  TRANSFORMATIONS
```

TRUTH TABLES BECOME ALGEBRAIC

Truth tables are used to determine the truth value of a complex expression in PC. Basically, the truth table method exhaustively surveys every possible value of every variable in order to determine the truth value of a complex expression. The algorithm is O(2^n), where n is the number of distinct variables. Three results are possible:

1. *Tautology*: the expression is true for all variable values,

2. *Contradiction*: the expression is always false, or

3. *Contingent*: the value of the expression depends upon specific assignments of truth values to variables.

We will illustrate the LoF algebraic technique for determination of the truth value of a complex expression for each of these cases.

Evaluation of Complex Expressions

We illustrate two different techniques that are functionally equivalent to the truth table method. The algebraic method uses the transformation rules of LoF to convert an expression to its simplest form. The simplest form of a tautology is ( ) , which is attached to the value TRUE; the simplest form of a contradiction is >< , which is attached to the value FALSE. We include one example of the arithmetic method, using the theorem of the BRIDGE. The Bridge has been used previously to demonstrate that the axiom of Distribution is a tautology.

Example I: algebraic simplification of a tautology

Consider the PC expression:

```
  ((a -> b) & (b -> c)) -> (a -> c)       Transitivity of Implication
```

Transcribe into LoF:

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

And simplify by the transformation rules:

```
      ==>       ((a) b) ((b) c)     (a) c                  Involution
      ==>       (    b) ((b) c)     (a) c                  Pervasion (a)
      ==>       (b    ) (    c)     (a) c                  Pervasion (b)
      ==>       (b    ) (     )     (a) c                  Pervasion  c
      ==>               (     )                            Dominion
```

Finally, transcribe the result back into PC:

```
            TRUE.  The expression is a tautology.
```

The algebraic technique is a simple and direct exercise in pattern-matching. We include all the steps, with explanation. Due to the simpleness of the rules, convergence is invariably rapid.

Example II: algebraic simplification of a contradiction

Consider the PC expression:

```
  (a -> (b -> c)) = ¬(b -> (a -> c))      Negated  Law  of  Permutation
```

Transcribe into LoF:

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

Since LoF is an equational system, we can transcribe and simplify each side independently. The transcription is of the form

```
                  E  =  (E)              E = (a) (b) c
```

which is the equational form of contradiction. It is permissible to disregard the order of the three expressions that constitute "E" in LoF just as it is permissible to disregard their order when assigning the numerosity "3" to them in integer arithmetic. For demonstration purposes only, we will evaluate the same expression by converting it to a non-equational form using the parens definition of logical equivalence. *Rhs* is right-hand-side; *lhs* is left-hand-side.

Equivalence

```
            rhs = lhs  <==>  ( ( ( rhs ) lhs ) ( ( lhs ) rhs ) )
```

Transcribe into LoF:

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

Simplify:

```
  ==> ( ( ((a)(b) c)           ) ( ( ((b)(a) c) ) (a)(b) c ) )  Replication
  ==> (    (a)(b) c              (    (b)(a) c    (a)(b) c ) )  Involution
  ==> (    (a)(b) c              (           ><            ) )  Pervasive
  ==> (                          (                         ) )  Dominion
  ==>                          ><                               Involution
```

Transcribe back into PC:

```
                    FALSE.  The expression is a contradiction.
```

The transformation rules were used here with complex expressions, rather than the simple expressions in the last example. With more, somewhat redundant steps, the same conclusion can be reached by manipulating only simple expressions.

Example III: algebraic simplification of a contingent expression

Consider the PC expression:

```
      (a V b) & (a -> b)
```

Transcription:

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

Simplification:

```
                    ==>  b ( (a  ) ((a)  ) )                   Distribution
                    ==>  b ( (a  ) (     ) )                   Pervasion (a)
                    ==>  b (       (     ) )                   Dominion
                    ==>  b         ><                          Involution
```

Transcribe back into PC:

```
                    CONTINGENT on the value of b.
```

In fact, the expression has been shown to be equivalent to b,

```
      (a V b) & (a -> b) = b
```

Example IV: arithmetic evaluation of a contingency

Consider the expression in the prior example:

```
            (a V b) & (a -> b)
```

with the transcription:

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

Bridge on the token a, by substituting each of its possible ground values:

Let *a* = ><

```
                  ( ( a  b) (( a ) b) )            Expression
             ==>  ( ( >< b) ((>< ) b) )            Substitute
             ==>  ( (    b) ((   )  ) )            Dominion
             ==>  ( (    b)           )            Involution
             ==>         b                         Involution
```

Now let *a* = ( )

```
                  ( ( a  b) (( a ) b) )            Expression
             ==>  ( (( ) b) ((( )) b) )            Substitute
             ==>  ( (( ) b) (      b) )            Involution
             ==>  ( (( )  ) (      b) )            Dominion
             ==>  (         (      b) )            Involution
             ==>                   b               Involution
```

Since the expression is equal to b for all possible values of the variable *a*, it is equal to b in all cases. Note that the reduction of ground states does not require the Pervasion rule.

THE LOF ALGEBRAIC DECISION PROCEDURE

The previous examples convey a taste of using the LoF formalism. Choice of transformation rules to apply at any given time is usually multiple and obvious. Application of a rule is simple and straight-forward. Convergence on the simplest form of an expression is rapid. And both algebraic and arithmetic techniques are available at all times.

LoF algebraic simplification is a decision procedure for PC that has very desirable algorithmic properties. It

1. converges rapidly,

2. is composed of relatively trivial algorithms,

3. is easily mechanized.

We will next show that the LoF formalism introduces the same profound simplification into the proof theory of PC.

Inference Rules Become Algebraic

LoF makes no distinction between the transformation of expressions and the proof of consequences from groups of expressions. Both cases are equational systems: transformations applying to single equations, proofs to systems of equations.

In this Section we show that the method of proofs in PC is equivalent to the solution of simultaneous Boolean equations. Like the set of transformations in PC, the set of inference rules in PC collapses to a smaller set of algebraic transformations in LoF. Many of the distinctions made as different methods of proof become one. More fundamentally, the algorithmic solution of systems of LoF equations provides a decision method for the proof theory of PC by dispensing with the distinction between transformation and proof.

The traditional argument for the necessity of a proof procedure in addition to the truth table method is the intractability of truth tables for large problems. As has been demonstrated, this problem is solved within LoF. The difficulty with proof procedures is that they often require ingenuity on the part of the mathematician in the selection and sequencing of the inference rules. The LoF algebraic approach corrects both the intractability of truth tables and the obscurity of inference rules.

We first demonstrate the reducibility of inference rules to the LoF algebra. Then we provide examples of LoF techniques for the proof of theorems.

Inference Rules

We divide the traditional inference rules of PC into three categories:

1. Access rules:
   Conjunction, Simplification, Addition.

2. Irrelevant inference rules:
   Modus Ponens, Modus Tollens, Disjunctive Syllogism.

3. Syllogistic rules:
   Hypothetical Syllogism and Dilemma.

In addition, we will address a selection of proof strategies:
Conditional Proof, Indirect Proof, and Resolution.

By transcribing these inference rules and proof procedures into LoF, we demonstrate that they reduce to a small number of algebraic operations that are not essentially different that those introduced by the transcription of PC transformation rules into LoF. The new ingredient is the concept of ACCESS. Some inference rules are restricted in application to entering or exiting the syntactic manipulation phase of the algebraic proof process.

As well, the concept of Syllogism is shown to be a single abstract structure rather than many different types of inference.

Combination Rules and Access

The combination rules in PC essentially define the method by which separate premises are combined.

Conjunction

```
      a
      b
   -------
    a & b
```

A new representational convention is introduced in the proof theory of PC: separate premises are written on separate lines. The underline represents the step of combining the premises to form the conclusion written below. An inference rule declares that the form of the combination is valid in PC. As well, most inference rules are binary in scope.

An alternative representation in PC of an inference rule is:

```
premise & premise => conclusion
```

where => is logical implication (as opposed to -> which is material implication). Logical implication assumes that the premises are valid, whereas material implication does not.

Applied to the Rule of Conjunction, this representation reads:

```
a    &    b    =>  a & b
```

The definitional nature of the Rule of Conjunction is obvious in this format.

The "premise-underline-conclusion" representation is an unnecessary complexity in LoF, which is an equational rather than implicational system. Instead of using a representation that permits the construction of a conjunctive fact from two individual facts, all facts to be considered in a LoF proof can be stored directly in a conjunctive structure. That is, both

```
a
b
```

and

```
a & b
```

are represented as

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

The rule of Conjunction becomes a Rule of ACCESS:

> Before any syntactic manipulations are performed, separate premises that are assumed TRUE can be combined by LoF conjunction: ((a)(b)(c) ...)
>
> After all syntactic manipulations are completed, expressions combined by LoF conjunction may be extracted as separate TRUE conclusions.

The latter half of the Rule of Access is a transcription of the PC inference rule of Simplification:

```
                 a & b                        a & b
                -------          and         -------
                   a                             b
```

Access has the advantage of partitioning the combination of premises and the extraction of conclusions from the syntactic manipulation that constitutes the action of proof.

To demonstrate the connection between the PC rules of Conjunction and Simplification to LoF, we will transcribe both rules and simplify them:

Conjunction

```
                a & b -> (a & b)
```

Transcribe and simplify:

```
                (((a)(b))) ((a)(b))
           ==>  (  ><    ) ((a)(b))              Pervasion ((a)(b))
           ==>  (        )                       Dominion
```

This trivial example points out the LoF transformation rules that correspond to the intent of the "premise-underline-conclusion" notation of PC. The relationship of Pervasion that exists between the premises and the conclusion must be complete (that is, all of the content of the formal space must be extracted) for Dominion to assure that the expression reduces to ( ), or its attachment, TRUE. An inference rule is that arrangement of terms such that the conclusion is completely Pervasive over the premises. This configuration is present in all PC inference rules, thus it can be identified as the fundamental characteristic of an inference rule.

Addition At Exit

The PC rule of Addition is also restricted in application to exiting the syntactic evaluation. Addition,

```
                    a
                 -------
                  a V b
```

becomes ADDITION AT EXIT:

> After all syntactic manipulations are completed, any token may be added to the extracted results by disjunction.

The LoF structure of Addition follows:

Addition

```
                 a -> a V b
```

Transcribe:

```
                 (a) a b
            ==>  ( ) a b                                 Pervasion a
            ==>  ( )                                     Dominion
```

The ability to add a disjunct to a true PC expression is equivalent to the Rule of Dominion in LoF. PC lacks a null concept such as <void>, but a close translation of Dominion into PC would be:

```
                 FALSE -> FALSE V a
```

Implication Rules

The four traditional inference rules in PC that use implication (conditionals) all reduce to LoF expressions that can be simplified or evaluated by the transformation rules of LoF, as can Disjunctive Syllogism.

The irrelevant rules of inference

The three irrelevant rules of inference all have the property that (seen from the LoF notation) their premises embody their conclusions. The conclusions can be extracted from the premises without the introduction of new or unique principles of calculation or inference.

Modus Ponens

```
                     a -> b
                       a
                    --------
                       b
```

The transcription and simplification of the expressions above the line is:

```
                    ( ((a) b) (a) )
                    ( (b    ) (a) )                          Pervasion (a)
```

The Rule of Access then permits the extraction of *b* as *a* result.

Alternatively the entire Modus Ponens expression could be transcribed into a single LoF expression. If this expression were simplified, it must necessarily lead to the value of ( ) , since, as an inference rule, it must be a tautology.

Represent Modus Ponens as

```
                    ( (a -> b) & a ) -> b
```

Transcribe and simplify:

```
                    ( ( ((a) b) (a) ) ) b
               ==>      ((a) b) (a)     b                    Involution
               ==>       ( >< ) (a)     b                    Pervasion (a) b
               ==>       (    )                              Dominion
```

Modus Ponens can be seen to have the characteristic Pervasion-Dominion pattern of inference rules, compounded with one level of complexity via Involution.

Modus Tollens

```
                     a -> b
                       ¬b
                    --------
                       ¬a
```

Transcription of the premises and their reduction:

```
                    ( ((a) b) ((b)) )
               ==>  ( ((a) b)   b   )                        Involution
               ==>  ( ((a)  )   b   )                        Pervasion b
               ==>  ( ((a)  ) ((b)) )                        Involution
```

Here, Involution has been applied to convert the simpler expression (a b) into a conjunctive form. From this form, either (a) or (b) can be extracted as conclusions by Access.

Analysis of the entire rule:

```
                  ((a -> b) & ¬b) -> ¬a
```

Transcription and simplification:

```
                  ( ( ((a) b) ((b)) ) ) (a)
             ==>      ((a) b)   b       (a)                 Involution twice
             ==>      (     )   b       (a)                 Pervasion b (a)
             ==>      (     )                               Dominion
```

This rule has the characteristic pattern of Pervasion-Dominion compounded with two applications of Involution.

Disjunctive Syllogism

```
                   a V b
                    ¬a
                  -------
                     b
```

Transcription and reduction of the premises:

```
                  ( (a b) ((a)) )
             ==>  ( (a b)   a   )                           Involution
             ==>  ( (  b)   a   )                           Pervasion a
             ==>  ( (  b) ((a)) )                           Involution
```

The conclusion *b* may be extracted by Access.

Analysis of the entire rule:

```
                  ((a V b) & ¬a) -> b
```

Transcription and simplification:

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

             ==>      (a b)   a       b                     Involution twice
             ==>      (   )   a       b                     Pervasion a b
             ==>      (   )                                 Dominion
```

Again we see that this rule reduces to the characteristic pattern compounded by two applications of Involution.

The syllogistic rules

The following two rules exhibit a complexity of form of their premises that stops a direct evaluation of the conclusion.

Hypothetical Syllogism

```
                  a -> b
                  b -> c
                 --------
                  a -> c
```

This rule has been previously addressed in the Section on EVALUATION OF COMPLEX EXPRESSIONS. There it was called Transitivity of Implication.

Transcription of the premises:

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

The premises of Hypothetical Syllogism do not reduce to a form from which the conclusions may be extracted. Thus, it may be viewed as fundamental. By including the conclusion, we form the LoF prototype form for all syllogisms. Rather than there being 24 forms of syllogism, in LoF they reduce to one form:

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

The 24 traditional forms can be seen to arise from this prototype by permuting the order of the three sentences (a factor of 6) and by negating (bounding, putting a parens around) each variable (a factor of 4).

Further simplification of the form of SYLLOGISM pin-points the source of its uniqueness:

```
            ==>      (    b) ((b) c)     (a) c             Pervasion (a)
            ==>      (    b) ((b)  )     (a) c             Pervasion c
            ==>      (    b) (     )     (a) c             Pervasion (b)
            ==>              (     )                       Dominion
```

We see that three applications of Pervasion are necessary, in distinct contrast to the previous three rules. This distribution of the parts of the conclusion across separate premises is the fundamental characteristic of all syllogistic forms that differentiates them from the other conditional inference rules.

Dilemma

```
                  a -> b
                  c -> d
                  a V c
                 --------
                  b V d
```

Transcription of the premises:

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

Like Hypothetical Syllogism, the premises of Dilemma do not reduce to a form from which the conclusion can be extracted. It remains to be seen whether or not Dilemma incorporates structural features that are different from those of the Syllogism.

Analysis of the entire rule:

```
           ( (a -> b) & (c -> d) & (a V c) ) -> (b V d)
```

Transcription and simplification:

```
           ( ( ((a) b) ((c) d) (a  c) ) ) b d
      ==>      ((a) b) ((c) d) (a  c)     b d              Involution
      ==>      ((a)  ) ((c) d) (a  c)     b d              Pervasion b
      ==>      ((a)  ) ((c)  ) (a  c)     b d              Pervasion d
      ==>        a       c     (a  c)     b d              Involution twice
      ==>        a       c     ( >< )     b d              Pervasion a c
      ==>                      (    )                      Dominion
```

Thus, we see that Dilemma has characteristics of Hypothetical Syllogism compounded with Involution. Dilemma then is a complex form of Syllogism, but is not inherently of a different class.

Summary of the Analysis of Inference Rules

In the proceeding Section, we analyzed the rules of inference from the perspective of LoF. Three types of rules emerged:

1. Access rules (Conjunction, Simplification, and Addition) that permit the selection of specific parts of a LoF structure upon entering or exiting the syntactic manipulations.

2. Irrelevant rules (Modus Ponens, Modus Tollens, and Disjunctive Syllogism) that incorporate the conclusion directly into the structure of the premises.

3. Syllogisms (Hypothetical Syllogism and Dilemma) that distribute components of the conclusion across different premises.

There are only two new principles introduced by PC inference rules and the methods of proof, those of ACCESS and SYLLOGISM. The apparent diversity of rules in PC is created by modifying these abstract principles with a number of applications of Involution.

The power and abstraction of the LoF formalism has permitted us to condense a bewildering variety of inference rules (that have evolved rather arbitrarily over hundreds of years) into a concise set of logical functionalities. To emphasize the utility of the LoF analysis of inference, we include an example in which we use the characteristic LoF form to generate a new inference rule for PC.

Generation of an Inference Rule from LoF Templates

Reversing the steps of simplification of an inference rule, we can generate any number of inference rules.

Basis:

```
                  ( )
```

Construction:

```
                    (     )                               True basis
            <==     (     )     b      (a)                Dominion
            <==     (    b)     b      (a)                Pervasion b
            <==     (    b) (   (b))   (a)                Involution
            <==     ((a) b) ((a)(b))   (a)                Pervasion (a)
            <==  (( ((a) b) ((a)(b)) )) (a)               Involution
```

Transcribe into PC:

```
                  ( (a -> b) & (a -> ¬b) ) -> ¬a
```

or:

```
                   a ->  b
                   a -> ¬b
                  ---------
                      ¬a
```

We have constructed an inference rule for the proof strategy called Indirect Proof.

Proof Strategies

We will now demonstrate that the proof strategies of Conditional Proof, Indirect Proof, and Resolution are not essentially different from inference rules or logical transformations. That is, the additional machinery introduced by these techniques is actually irrelevant.

Both Conditional Proof and Indirect Proof introduce the idea of assumptions that are assumed and then discharged in place of the concept of premises as a basis of proof. Resolution seeks proof by including the negation of the conclusion and proving an inconsistency.

The power provided by an algebraic system such as LoF is that the pattern rules apply to arbitrary expressions. We have made the distinction previously between small letter variables and capital letter variables. Small letter variables stand in place of either ground value; capital letter variables stand in place of any arbitrary parens expression. In logic and in proof theory, a similar distinction exists between inference rules and proof strategies. The central idea is that due to the nature of algebraic substitution, the premise/inference/conclusion structure does not need to be differentiated from the logic-rule/substitution structure. We are therefore free to use the algebraic techniques above to address proof strategies. The concept of STEPS in logic proof is replaced by successive substitutions in LoF.

Conditional Proof

The schema for conditional proof is:

```
              a                    Assumed
              b                    Prove by steps from a
          --------
           a -> b
```

The idea is that if one can prove *b* by assuming *a*, then one has proved that *a implies b*. We will show that this machinery adds nothing within the context of the LoF formalism.

Analysis of the schema:

```
          (a & b) -> (a -> b)
```

Transcribe and simplify:

```
          ( ((a)(b)) ) (a) b
   ==>       (a)(b)    (a) b                 Involution
   ==>       (a)( )    (a) b                 Pervasion of b
   ==>          ( )                          Dominion
```

We see the characteristic signature of an inference rule: Pervasion-Dominion compounded by an application of Involution. Conditional proof is a necessary tool only because the PC formalism is unnecessarily complex.

Indirect Proof

The schema for Indirect Proof is:

```
          a                           Assumed
       b & ¬b                         Prove by steps from a
      --------
         ¬a
```

The idea is that if one can prove a contradiction by assuming a, then the negation of a must in fact be TRUE.

Analysis of the schema:

```
      (a & (b & ¬b)) -> ¬a
```

Transcribe and simplify:

```
          ( ((a) ( ((b) ((b))) )) ) (a)
   ==>       (a)    (b)   b         (a)          Involution thrice
   ==>       (a)    ( )   b         (a)          Pervasion b
   ==>              ( )                          Dominion
```

Indirect Proof, like Conditional Proof, follows the characteristic pattern for all inference rules. It, too, is subsumed by LoF.

Resolution

The schema for Resolution is:

```
            x                         Any premise
            y                         Any premise
       ¬conclusion                    The conclusion negated
      -------------
      contradiction
```

The idea is that if the premises and the negated conclusion are joined by conjunction, then this complex expression will yield a contradiction only if the conclusion is TRUE.

Since the schema for Resolution is abstract, we will instantiate it with a specific example (from the inference schemata of the Stoics, c. 200 B.C.).

Consider:

```
                  (a & b) -> c
                       ¬c
                        a
                 --------------
                       ¬b
```

In the format of Resolution:

```
                   (a & b) -> c
                       ¬c
                        a
                        b                         Negated conclusion
                 ---------------
                  contradiction
```

Rewrite:

```
                 ((a & b) -> c) & ¬c & a & b) -> FALSE
```

Transcribe and simplify:

```
           ( ( ( (((a)(b))) c ) ((c)) (a) (b) ) ) ><
      ==>      (   (a)(b)   c )   c   (a) (b)              Involution thrice
      ==>      (       ><     )   c   (a) (b)              Pervasion c (a)(b)
      ==>  ( )                     Dominion
```

The characteristic signature again appears for Resolution.

Note that, as it is stated, the conclusion that the four premises lead to a FALSE conclusion is itself TRUE. Alternatively, we could have analyzed only the premises, in this form:

```
             ( ( (((a)(b))) c ) ((c)) (a) (b) )
     = =>    ( (   (a)(b)   c )   c   (a) (b) )            Involution twice
      ==>    ( (      ><      )   c   (a) (b) )            Pervasion c (a)(b)
      ==>    ( (              )               )            Dominion
      ==>             ><                                   Involution
```

Here the set of premises embody a contradiction. Generally, only the premises (with the negated conclusion) need be evaluated in Resolution since the implied conclusion is empty.

Summary of Proof Techniques from the Perspective of LoF

We have analyzed exhaustively the transformations, inference rules and proof procedures in PC in order to demonstrate that, from the perspective of LoF, they are unnecessarily complex. LoF provides a complete and consistent axiomatization that is morphic with PC, easy to understand, and simple to use.

Transformation Rules

((a))	=	a	INVOLUTION
a a	=	a	REPLICATION
a ( )	=	( )	DOMINION
a (a b)	=	a (b)	PERVASION
a ((b)(c))	=	((a b)(a c))	DISTRIBUTION

Access Rules

Before transformation

Combine all desired premises by LoF conjunction: ((a)(b)(c) ...)

After transformation

Extract all desired conclusions from LoF conjunction.
Add all desired tokens by disjunction.

An Example of LoF Proof

We will now focus on a single logical problem, demonstrating the different LoF techniques that lead to its solution. For purposes of comparison, we include the proof in PC. The four methods of proof in LoF include two Resolution techniques (one using the algebra and one using the arithmetic) and two direct solution techniques (again one algebraic and one arithmetic).

Consider this problem:

1. Either John is the murderer or he is not the burglar,
   but only if he's telling the truth,
   unless, of course, Bob is lying.

2. Neither Bob is lying nor does John have a gun.

3. If John is either the driver or the burglar,
   he is the murderer.

4. Is John telling the truth?

To convert this problem to symbolic form, let:

M	=	John is the Murderer
B	=	John is the Burglar
T	=	John is telling the Truth
L	=	Bob is Lying
G	=	John has a Gun
D	=	John is the Driver

The problem is then symbolized in PC as follows:

1.	$(M \vee \neg B) \rightarrow (\neg L \rightarrow T)$	Premise 1
2.	$\neg(L \vee G)$	Premise 2
3.	$(D \vee B) \rightarrow M$	Premise 3
4.	T	Conclusion

Since our interest is in the symbolic processes of proof, and not in the conversion of English into PC, from now on we will discuss only the symbolic representation above.

Solution in PC:

```
 1.  (M V ¬B) -> (¬L -> T)         Premise
 2. ¬(L V G)                        Premise
 3.  (D V B) -> M                   Premise

 4.   B                             Conditional proof, assume 4.
 5.   B V D                         Addition to 4.
 6.   D V B                         Commutativity of V
 7.   M                             Modus Ponens, 3. and 6.
 8.   B -> M                        Conditional proof, 4. to 7.
 9.  ¬B V M                         Conditional exchange, 8.
10.   M V ¬B                        Commutativity of V
11.  ¬L -> T                        Modus Ponens, 1. and 10.
12.  ¬L & ¬G                        DeMorgan, 2.
13.  ¬L                             Simplify, 12.
14.        T                        Modus Ponens, 11. and 13.
```

Therefore, John is telling the Truth.

The outstanding characteristic of this proof is the insightful beginning assumption of B so that the conditional sub-proof (steps 4 to 7) could provide the basis for the straight-forward steps that follow (steps 9 to 14). We will see in the LoF proofs that follow that the process requires no insight, it is entirely algorithmic.

Techniques for the Proof of the Example in LoF

We will demonstrate two different techniques of proof in LoF: an algebraic resolution and an arithmetic resolution.

Both LoF techniques will use the following transcription of the problem:

```
1.  (M V ¬B) -> (¬L -> T)  ==>  (M (B)) ((L)) T
                           ==>  (M (B))   L   T           Involution

2.  ¬(L V G)               ==>  (L G)

3.  (D V B) -> M           ==>  (D B) M

4.  T                      ==>   T
```

Combination of the premises and the negated conclusion by LoF conjunction:

```
(   ((M (B)) L T)   ((L G))    ((D B) M)   ((T))    )
(   ((M (B)) L T)     L G      ((D B) M)     T      )   Involution
```

Algebraic resolution

Technique

1. Join the premises and the *negated* conclusion by conjunction.

2. Apply transformation rules to simplify.

3. If the result is ><, then the conclusion is TRUE.

Example

```
         ( ((M (B)) L T) L G ((D B) M) T )
    ==>  ( ((M (B))    ) L G ((D B) M) T )        Pervasion L T
    ==>  (   M (B)       L G ((D B) M) T )        Involution
    ==>  (   M (B)       L G ((D B)  ) T )        Pervasion M
    ==>  (   M (B)       L G   D B     T )        Involution
    ==>  (   M ( )       L G   D B     T )        Pervasion B
    ==>  (     ( )                     )          Dominion
    ==>        ><                                 Involution
```

The conclusion, T, is TRUE.

Arithmetic resolution

Technique

1. Join premises and negated conclusion by conjunction.

2. Assign values, either ( ) or >< , to variables such that the expression does not evaluate to FALSE. That is, choose values that avoid a contradiction.

3. If a contradiction cannot be avoided, then in all cases the conclusion must be TRUE.

Example

```
         ( ((M (B)) L T) L G ((D B) M) T )
```

If a variable standing alone at depth = 1 is assigned ( ) , then its value will dominate the expression, forcing it to >< . Therefore, all variables standing alone at the first level of depth must be assigned ><, the value that may propagate an unknown change through the system. In this case, the variables L, G, and T can be assigned the value >< :

```
          ( ((M ( B)) L  T ) L  G  ((D B) M  ) T  )
     ==>  ( ((M ( B)) >< ><) >< >< ((D B) M  ) >< )           L G T  = ><
     ==>  (   M ( B)              ((D B) M  )    )           Involution
     ==>  (  >< ( B)              ((D B) >< )    )           M = ><
     ==>  (     ( B)                D B          )           Involution
     ==>  (     (><)               >< ><         )           D B = ><
     ==>         ><
```

A contradiction was unavoidable, therefore the conclusion is TRUE.

The above arithmetic technique is also an algorithm for generating counter-examples to refutable proofs.

Direct algebraic solution

It is not necessary to include the conclusion explicitly, since all valid conclusions are incorporated directly in the premises. The following technique is more complex, but demonstrates that the conclusion, T, can be extracted form the premises alone. The steps of the extraction indicate the form of the relational dependencies between the premises.

Technique

1. Combine the premises in conjunction.

2. Apply simplifying transformations.

3. If the desired conclusion is extractable, do so; otherwise decrease the depth of nested parentheses until tokens for the desired conclusion are either extractable or distributed. The configuration of the distribution specifies the dependencies of the conclusion yet to be filled if the conclusion is to be valid.

Example

```
          ( ((M (B)) L T) L G ((D B) M)   )              Premises
     ==>  ( ((M (B))   T) L G ((D B) M)   )              Pervasive L
```

Note that the conclusion T is not included within the structure of the example. T is, necessarily, mentioned somewhere in the Premises.

The expression apparently does not simplify further, so the tecnique is to transform it so that the conclusion, T, is at a shallower depth. We are. in effect, digging the conclusion out. In order to clearly show the terms of the next transformation, we isolate them in text-space:

```
      ==>  (((   M(B)   )   T      )          LG((DB)M)   )
      ==>    (   M(B)   LG((DB)M)  )   ((T)   LG((DB)M)   )       Depth
```

The cost of the Depth transformation is that it replicates other premises. However, the two resulting expressions can be simplified independently, by the Rule that all forms in a void-space are independently void-equivalent:

```
      ==>  (M (B) L G ((D B) M))    ((T) L G ((D B) M))           Reformat
```

Expression I

```
                 ( M (B) L G ((D B) M) )
             ==>  ( M (B) L G ((D B)  ) )                        Pervasive M
             ==>  ( M (B) L G   D B     )                        Involution
             ==>  ( M ( ) L G   D B     )                        Pervasive B
             ==>  (   ( )               )                        Dominion
             ==>       ><                                        Involution
```

Expression I, which does not incorporate the variable T, must vanish for T to avoid dependencies. The goal of applying the Depth Rule was to partition the conclusion T into one expression, making the others void-equivalent.

Expression II

```
                 ( (T)   L G   ((D B) M) )
             ==>  ( (T) ((L G)) ((D B) M) )                      Conjunctive form
             ==>     T                                           Extraction
```

Note that the remaining terms in the conjunctive form are also TRUE. In fact, (L G) and (D B) M are both original premises.

The algebraic transformations that isolate T as a conjunct can be justified structurally as well. Here is the direct algebraic solution, achieved by isolating the desired variable within a single conjunct. The direct solution is:

```
            ( ((M (B)) L T) L G ((D B) M)   )               Premises
            ( ((M (B))   T) L G ((D B) M)   )               Pervasion L
            (( (M (B)((D B) M)) ((T)((D B) M)))   L G  )    Distrib ((D B) M)
            (( (M (B)(       )) ((T)((D B) M)))   L G  )    Pervasion M (B)
            ((                  ((T) ((D B) M)) ) L G  )    Occlusion
            (                    (T) ((D B) M)    L G  )    Involution

            [                    [T] [(D B) M]  [[L G]])    Conjugate form
```

From the conjugate form, select T.

Boolean linear algebra

Since LoF is equational, each premise can be expressed as a Boolean equation. If the simultaneous solution of these equations yields the desired conclusion, then the conclusion is TRUE. This technique is useful only because of the deep simplification that takes place when PC is transcribed into LoF.

Technique

1. Express premises as equations, each equal to ( ) .

2. Solve individual equations for values of individual variables.

   Case I: (x y) = ( )

   implies x and y must be FALSE or >< .

   Case II: x y = ( )

   branches into two possibilities, x = ( ) and y = ( ) . If both possibilities yield the conclusion, then the conclusion is TRUE in all cases.

3. Substitute values into remaining equations until the value of the conclusion is isolated.

Example

```
          (M (B)) L T  =  ( )                  Premise 1

               (L G)   =  ( )                  Premise 2

            (D B) M    =  ( )                  Premise 3
```

From Premise 2,

```
          (L G)  =  ( )
           L G   =   ><
```

Therefore, L = >< and G = ><

Void-substituting known values into the other two premises yields a new Premise 1:

```
          (M (B)) >< T  =  ( )                 Premise 1'
```

Premise 3 forms a branch, *either* form in the shallowest space may equal ( ):

```
                  (D B) M  =  ( )
```

Case I:

```
            (D B)  =  ( )

      Therefore, D = >< and B = ><

      Substituting into 1':

            (M (><)) T = ( )
       ==>           T = ( )                  Occlusion
```

Case II:

```
            M = ( )

      Substituting into 1':

            ( ( ) (B)) T = ( )
       ==>             T = ( )                Occlusion
```

In both cases T is TRUE, therefore it is a valid conclusion.

THE MAPPING ONTO PREDICATE CALCULUS

Propositional Calculus is a sub-language of First-Order Predicate Calculus (FOPC). Therefore the LoF techniques previously discussed apply directly to FOPC. FOPC extends PC by including relations, functions and quantifiers. We offer no innovations for the representation of relations or functions in this Section. (Although LoF does have unique representations for set theoretic relations such as MEMBER and number theoretic functions such as ADDITION, these extensions will not be discussed here.) The LoF treatment of quantification follows.

LoF incorporates quantification of variables without introducing separate tokens for the Universal and Existential quantifiers. The parens notation, surprisingly, also accommodates both of these concepts.

We do not need to analyze the theory of quantification to the same extent as we did, say, proof theory in PC, since quantification is largely reinterpreted for automated reasoning. Specifically automated systems never entertain infinite collections. Universal quantification means all of a specific finite set; existential quantification means at least one of a finite set. Thus, quantifiers are actually interpreted as search routines for finite databases. This characteristic will be expanded in the Section on the Losp programming language.

For finite collections, Universal quantification can be treated as the conjunction of all instances quantified over. Existential quantification can be viewed as the disjunction of all instances quantified over. These relationships are the key to understanding LoF quantification.

LoF QUANTIFICATION

We introduce a set of symbols that represent sets of objects or relations:

$$\{A, B, C, \dots\}$$

When one of these symbols is used, it will be followed immediately by the instance of the set. We will use

$$\{a, b, c, \dots\}$$

to represent specific instances, and

$$\{x, y, z, \dots\}$$

to represent quantification variables. Therefore, read the token *Ax* as a single symbol.

The mapping of FOPC quantification onto LoF follows:

Concept	FOPC Notation	LoF Notation
If it is an x, then it is an A.		
All x are A	(All x) Ax	(x) Ax
If it is an x, then it is not an A.		
No x is A	(All x) ¬Ax	(x)(Ax)
Not all x are A.		
Some x are not A	(Exist x) ¬Ax	((x) Ax)
It is not the case that no x is A.		
Some x are A	(Exist x) Ax	((x)(Ax))

MAPPING OF QUANTIFIERS ONTO LOF

We indicate the FOPC quantifiers with the tokens All and Exist for typographical convenience. The concept description above is phrased so that the LoF representation is easy to understand. Specifically, the FOPC concept All can be read as "x implies A": if it is an x, then it is an A. The concept Exists can be read as the negation of "x implies not A": it is not the case that no x is A.

It must be noted that the LoF expression ((x) (Ax)) is not used in the same sense as the expression for conjunction, ((a)(b)) , even though the parens configuration is identical. The difference is the assigned meaning of the token x , the variable of quantification.

Quantifier Negation Equivalencies

The FOPC equivalencies between negated quantified expressions are all instances of an simple expression compounded by one or two applications of Involution in LoF:

FOPC Equivalence Rule	LoF Transcription
¬(Exist x) Ax = (All x) ¬Ax	(((x) (Ax))) = (x) (Ax)
¬(All x) Ax = (Exist x) ¬Ax	((x) Ax) = ((x) ((Ax)))
¬(Exist x) ¬Ax = (All x) Ax	(((x) ((Ax)))) = (x) Ax
¬(All x) ¬Ax = (Exist x) Ax	((x) (Ax)) = ((x) (Ax))

QUANTIFIER NEGATION

Due to the power of the Involution transformation, all the quantifier negation equivalence rules are simple applications of Involution in LoF.

LoF Categorical Propositions

The LoF quantifier mapping can be easily extended to include categorical relations and their equivalencies.

Type	Description	FOPC Formula	LoF Transcription
A	All A is B	(All x)(Ax -> Bx)	(x) (Ax) Bx
E	No A is B	(All x)(Ax -> ¬Bx)	(x) (Ax) (Bx)
O	Some A is not B	(Exist x)(Ax & ¬Bx)	((x) (((Ax) ((Bx)))))
I	Some A is B	(Exist x)(Ax & Bx)	((x) (((Ax) (Bx))))

CATEGORICAL TYPES MAPPED ONTO LOF

Categorical Quantifier Negation Equivalencies

The classical Square of Opposition in FOPC, and its associated equivalence relations is quite simple within LoF:

Type   ¬A = O

```
      FOPC        ¬(All x)( Ax -> Bx )  =  (Exist x)(   Ax & ¬Bx  )

      LoF         ((    x)(Ax)   Bx )   =  ((     x)( ((Ax) ((Bx))) ))
             ==>  ((    x)(Ax)   Bx )   =  ((     x)   (Ax)   Bx     )
```

Type   ¬I = E

```
      FOPC        ¬(Exist x)(   Ax & Bx  )    =  (All x) (Ax -> ¬Bx )

      LoF         (((     x)( ((Ax)(Bx)) )))  =  (    x) (Ax)   (Bx)
             ==>    (     x)   (Ax)(Bx)       =  (    x) (Ax)   (Bx)
```

Type   ¬E = I

```
      FOPC        ¬(All x)( Ax -> ¬Bx )  =  (Exist x) (  Ax & Bx )

      LoF         ((    x) (Ax)   (Bx))  =  ((     x) (((Ax) (Bx))) )
             ==>  ((    x) (Ax)   (Bx))  =  ((     x)   (Ax) (Bx)   )
```

Type   ¬O = A

```
      FOPC        ¬(Exist x)(   Ax & ¬Bx   )      =  (All x)( Ax -> Bx)

      LoF         (((     x) (((Ax) ((Bx)))) ))  =  (    x) (Ax)   Bx
             ==>    (     x)   (Ax)   Bx         =  (    x) (Ax)   Bx
```

Again we note that an originally bewildering set of equivalencies between the logical operators and quantifiers is reduced to simple expressions in LoF compounded with one or more application of Involution. Given the LoF rule of Involution, all the categorical quantifier negation relations become irrelevant.

Quantifier   Transformation   Rules

Since it is desirable to be able to translate quantified logical formulae into LoF, the following table of quantifier transformation rules is included:

```
===========================================================================

 Quantifier  Transformations                   LoF  Transcription

(All x)(p V Ax) = p V (All x)(Ax)

                                             (x)  p  Ax =  p  (x) Ax

(All x)(p -> Ax) = p -> (All x)(Ax)

                                             (x) (p) Ax = (p) (x) Ax

(All x)(Ax -> p) = (Exist x)(Ax) -> p

                                             (x) (Ax) p = (((x) (Ax))) p
                                      ==>    (x) (Ax) p =   (x) (Ax)   p

(Exist x)(p & Ax) = p & (Exist x)(Ax)

                                      ((x) (((p)(Ax)))) = ((p) (((x)(Ax))))
                                 ==>  ((x)   (p) (Ax) ) = ((p)   (x)(Ax)  )

(All x)(p & Ax) = p & (All x)(Ax)

                                        (x) ((p) (Ax))  = ((p) ((x) Ax ))

(Exist x)(p -> Ax) = p -> (Exist x)(Ax)

                                       ((x) ((p)  Ax )) =  (p) ((x)(Ax))

(Exist x)(p V Ax) = p V (Exist x)(Ax)

                                       ((x) ( p   Ax )) =   p  ((x)(Ax))

(Exist x)(Ax -> p) = (All x)(Ax) -> p

                                       ((x) ((Ax) p  )) =  (x) (Ax) p

(Exist x)(Ax V Bx) = (Exist x)(Ax) V (Exist y)(By)

                                          ((x) (Ax Bx)) = ((x)(Ax)) ((y)(By))

(All x)(Ax & Bx) = (All x)(Ax) & (All y)(By)

                                         (x) ((Ax)(Bx)) = (((x) Ax) ((y) By))

===========================================================================
                      QUANTIFIER  TRANSFORMATION   RULES
```

Proofs with Quantifiers

Four additional inference rules are introduced into FOPC by the existence of quantifiers. Each of these is expressed succinctly in LoF.

```
                Tableau  notation                Textual  notation
```

Universal Instantiation

```
                    (All x) Ax
                    ------------                 (All x) Ax -> Aa
                         Aa
```

Transcription ((x) Ax) Aa

This rule restricts the application of Pervasion when there is a quantifier variable within the formal space.

Existential Generalization

```
                          Aa
                    --------------               Aa -> (Exist x) Ax
                     (Exist x) Ax
```

Transcription (Aa) ((x) (Ax))

This rule can be seen to be identical to Universal Instantiation; the restriction of the use of Pervasion includes token within a parens.

Existential Instantiation

```
                     (Exist x) Ax
                    --------------               (Exist x) Ax -> Aa
                           Aa                         provided "a" is flagged
```

Transcription

```
                                                 (( (x) (Ax) )) Aa
                                            ==>     (x) (Ax)    Aa
```

Next we encounter a different kind of restriction:

Universal Generalization

```
                      a                            flagged
                     Aa                            by steps from flagged a
                ------------
                 (All x) Ax

or                                 (a & Aa) -> (All x) Ax      a is flagged

Transcription                      ( ((a) (Aa)) ) (x) Ax
                              ==>     (a) (Aa)    (x) Ax
```

Prenex Form

The LoF technique for converting a compound logical expression with quantifiers to prenex form is:

Remove all the tokens for quantification , eg (x) and place them in front of the expression. The remaining expression is of the appropriate form.

If a token of quantification was at an even depth, it is Universal. If the token was at an odd depth, it is Existential.

[[I had intended to continue this monograph with Skolemization, functions, and relations. Most of the intended topics are covered in subsequent separate monographs.]]