COMPUTATIONAL COMPLEXITY AND BOUNDARY LOGIC

William Bricken
January 2002

CONTENTS

SUMMARY

Logic has evolved over thousands of years within language. Mathematical logic is young, a creation of the 20th century. One of the most outstanding problems for mathematical logic is determining whether or not there exist tractable algorithms for exponential problems. An answer may lay in one of the simplest exponential problems, that of determining if a given logic expression is a tautology. Boundary logic (BL) provides a new set of computational tools which are geometric and algebraic, most definitely not conventional logic. These container-based tools are simpler than those of both mathematical and natural logic. Can boundary logic shed light on tautology identification?

After the nature of algorithmic complexity and BL are introduced, we explore conventional rules of inference and deduction from a BL perspective. Proofs of all but the distributive law are close to trivial using BL algorithms. Certainly none of the explicit structure of modern logic identifies complexity. We explore significantly difficult tautological problems which can be constructed using the rules of logic. None of these are complex, although some are non-trivial. It appears that compound logical rules do not identify complex problems, yet logical proof systems rapidly require exponential effort.

We then describe the central reduction algorithm of BL, called *virtual insertion*, and apply it to known tractable and intractable problems. The low-degree polynomial virtual insertion algorithm is not complete, however the tautologies it cannot reduce may be constrained to intractable problems. Thus, virtual insertion is an efficient decision procedure for tractable tautologies. This is valuable since almost all pragmatic problems are tractable. Using virtual insertion recursively produces a complete decision procedure for elementary logic, but one with the expected exponential complexity.

The boundary logic representations and algorithms described herein have been fully implemented in software and applied to computationally difficult practical problems, such as circuit design, tautology detection, and computer program optimization..

APPLICATIONS

The algorithms herein provide a novel and unique approach to any practical problems that can be expressed in the formal language of conventional logic, or Boolean algebra. We identify five general application areas:

1. ***Database querying and management*** (question answering)

Facts and database entries are expressed as logical assertions. Abstract relations between facts are expressed as rules with variables. To answer a query, derive the answer from the facts and the rules.

2. ***Program analysis and verification*** (compiling)

Program execution is expressed as logical formulas. To verify a program, derive the termination conditions from the execution formula.

3. ***Planning*** (state transformation and finite state machines)

States and state transformations are expressed as logical formulas. To transform the start into the goal, derive the final state from the initial state.

4. ***Circuit design*** (logic synthesis)

Circuits are expressed as logical formulas. To execute a circuit, substitute the Boolean input values and evaluate the formula. To minimize a circuit, apply transformations to the logical formula.

5. ***Decision making*** (decision analysis)

Decisions are expressed as Boolean choices, which are expressed as logical formulas. To make a complex network of decisions, evaluate the corresponding logical forms.

The set-insertion procedure provides the capability to derive, verify, transform, minimize, and evaluate logical problems. Some specific applications:

Determine the season schedule for a sports team.
Create weaving machine instructions for a particular pattern.
Reduce the logical complexity of a computer program.
Optimize the operational overhead and resource use of a schedule.
Determine the validity of a legal argument.
Transform an integrated circuit to fit a given hardware resource.
Determine a simple choice of actions for a complex decision sequence.
Validate a set of conclusions based on facts and rules.
Optimize an expert-system rulebase.

Identify redundancies in a sequence of processes.
Monitor railroad switches for potential collisions.
Manipulate binary information in a computer.
Develop a model of how children think.
Compute a result given a declarative program.
Arrange a meeting schedule for many busy executives.
Develop a plan of action when resources are limited.

INTRODUCTION

This section sets the stage by identifying the general terms and issues of algorithmic complexity. The style is non-mathematical, presenting the content in quite simple language. The objective is to identify the simplest problem which is too complex to be practical for computational algorithms.

Historical Context

During the 1960s, America rapidly built up its academic computational capabilities, helped by the Sputnik crisis, and by large investments from ARPA. In building essentially new departments of Computer Science, many schools drew from Electrical Engineering (thus there are many current Departments of Computer Science and Electrical Engineering), and many drew from Mathematics (thus there are many Departments of Mathematics and Computer Science). The professoriat which was installed in the 60s had a heavy theoretical bent; despite the admonitions of Dykstra and others familiar with programming, theoretical analysis of computational efficiency became a core subject. Since these first professors had enormous political influence, their predispositions still effect theoretical Computer Science today, even though computers themselves have changed radically.

Unfortunately complexity analysis suffers from two difficulties, lack of analytic tools and irrelevance.

Algorithmic complexity, the analysis of the efficiency of algorithms, found its basis in worst-case analysis. Basically, how well would an algorithm perform if an infinitely intelligent and devious mathematician made up problem cases for it? This problem turned out to conform to known tools, and still today it largely defines what we know about algorithmic performance. During the rapid build of expert systems in the early 1980s, researchers discovered a surprising invariant: practical problems, in particular pragmatic expert system rule-bases, were never the worst case. In fact, people don't think like devious mathematicians, and practical problems do not generate tangled and tortured data structures. This observation is strongly supported by the analysis of complex circuitry. Although the diversity of structures of logic networks, both combinational and sequential, is profound,

their functional variation is extremely limited. Like all pragmatic fields, the vast majority of possible structures are irrelevant.

Worst-case performance is vital for some limited application areas, in particular when one's life depends on the timely return of results, such as dynamic control of flight and emergency medical diagnosis. EDA has none of these constraints. However, it would be very beneficial if the computational complexity of EDA transformation tools was known. At least an engineer could know if the wait for results would take a minute or a year. Here again, algorithmic complexity is largely irrelevant. For all but the most simple cases of circuits (prevalent in the 1970s), all EDA algorithms are horribly complex. None are efficient.

There are several other *fundamental problems* with casting an evaluation of EDA tools into a framework of complexity analysis. For example:

- Complexity analysis applies to procedurally specified algorithms, those in which you tell the computer what to do step by step. Modern programming methods such as object-oriented programming are not procedural, although the same analyses may apply, they are not accessible without restructuring what is now known to be good programming practice.

- Complexity analysis applies to mathematical computational algorithms, not to the vast majority of pragmatic tools such as graphic user-interfaces, network switching, and design tools.

- Some algorithms take a huge amount of initialization effort and data structure design. The right way of stating a problem can often change its complexity.

- Some algorithms are not very complex, but what they have to do is. For example, imagine putting the Social Security numbers for all Americans in order. Sorting can be very efficient, but sorting millions of things may not be. Similarly, imagine gathering all financial transactions for a thousand people. Gathering can be very efficient, but gathering information from widely distributed, incompatible, and inaccessible sources may not be.

- Most importantly, algorithm analysis assumes and is built upon processing of linear strings, that is, typographical and stream structures. It bypasses far more powerful spatial and geometric analytic tools by historical accident, since early computers were based on processing linear and sequential streams of bits. Parallel computing, for example, has created a necessity for different types of analysis.

From this historical context, one finds that the first question and/or critique aimed at EDA design tools, most always from those of academic training and rarely from pragmatic application designers, is: what is the

computational complexity of this algorithm? Despite concerns for realism, BTC's Iconic Logic tools provide clear and powerful answers to this question.

Polynomial and Exponential

Complexity analysis asks the question: if an algorithm performs at some level given some input, what will happen to performance when the input is increased in size? If it takes, say 10 seconds to process 10 items, how long will 11 items take?

The answer to this question is expressed in terms of a mathematical equation which relates the effort expended by an algorithm to the number of things it processes. Let the number of things being processed be N. Basically there are two fundamental classes of mathematical complexity equations: polynomial and exponential (non-polynomial). *Polynomial* refers to a type of mathematical relation, one in which N is raised to some power, such as N^3 or N^1.5. *Exponential* refers to a relation in which some value is raised to the Nth power, such as 2^N. The following table shows why this distinction is important:

N	N^2	2^N
2	4	4
5	25	32
10	100	1024
20	400	>1000000
100	10000	>> atoms in the universe

Polynomial algorithms work reasonably well as the number of processed items increases. Exponential algorithms are not feasible for anything but very small numbers of items.

Some Terminology

NP: non-polynomial

non-polynomial: a relation which diverges extremely rapidly

polynomial: a relation which diverges reasonably, maintaining practicality

tractable: polynomial algorithms

intractable: non-polynomial algorithms

NP-complete: a class of NP problems which are all formally equivalent

heuristics: techniques which make the behavior of NP algorithms practical

SAT: a simple NP problem, asking, does this logic expression have a solution?

TAUT: the flipside of SAT, asking, does this problem have a non-solution?

SAT and TAUT

Historically, one of the simplest questions in logic is: is there a set of values which will make an expression *True*? Mathematically this is called the *Satisfiability Problem*, or SAT. A problem which is not satisfiable has no answers, you might say that it is a useless expression. This simple question, as it turns out, is also one of the most difficult to answer computationally. The issue is not how to determine an answer, that is well known. The issue is how much computational effort does it take to arrive at an answer. The state-of-the-art is that there are no known algorithms which are feasible for solving SAT problems with large N.

The ways in which uselessness can be hidden in logic expressions are difficult to identify. For example these logic expressions are not satisfiable:

A and (not A)

(A or B) and (A or (not B)) and ((not A) or B) and ((not A) or (not B))

((not (A or B)) or (A and B)) iff (not (((not A) or B) and ((not B) or A))

A sister problem to SAT is *tautology detection*, or TAUT. It asks: is there *any* set of values which would make an expression False? A tautological expression is always *True*, regardless of the values of its variables. Tautologies too provide no useful information about their variables. Negating any of the unsatisfiable expressions above makes them tautologies.

Fundamental research in the 1970s showed that many NP problems can be rephrased to be the SAT problem. To understand many problems, it thus suffices to analyze the behavior of algorithms for the SAT problem. There are some particular castings of SAT which are very valuable, one is Boolean minimization. Instead of identifying useless expressions, minimization looks for useless portions and islands within useful expressions. For example, the expression

A and (A or B)

could be restructured simply as

A

without changing its meaning. That is, the B variable is useless in this context, while both logical operations are irrelevant.

To turn Boolean minimization into SAT, one asks an exponential number of SAT questions, of the type: if some variables are held constant, will the remaining expression become totally useless? Above, one would discover that no matter what value variable B had, it would not effect the value of the entire expression. This does not necessarily identify a better way to structure the expression. Boolean minimization is not only intractable, it is "intractably intractable"; finding the minimal form of logic expressions requires an exponential number of exponentially difficult questions. Boolean minimization requires algebraic function analysis, not merely testing values.

Heuristics

How do any logic reduction algorithms work? They use *heuristics*, specific pieces of knowledge which limit the huge number of irrelevant structures, those structures which compute no useful functions. Heuristics put up stop signs for intractable algorithms, saying "don't bother to go here". Strong heuristics -- said another way, strong knowledge of the application domain -- reduce horrendous computational problems to approachable ones. Heuristics must at some point fail to help, otherwise they would always succeed in making difficult problems easy, that is, they would do away with difficulty.

The intractable nature of Boolean minimization, of SAT, and of TAUT cannot be made to go away. Intractability is an inescapable mathematical fact. However, it is possible to remove so much of the *apparent complexity* of common logical expressions, using tractable Boundary Logic (BL) means, that what remains is one of two things, either:

1. a pathological, devious structure, one which would very rarely appear in practical, applied problems, or

2. a very tangled structure for which further, exponential effort shows little pragmatic gain.

The inherently powerful container-based data structures of BL express logical problems in spatial geometric terms rather than in symbolic strings. In so doing, so much of the redundant clutter of conventional logical expressions, such as that exhibited in the examples above, disappears. What is left is the core of the functionality. This core can be exposed using tractable BL algorithms.

Satisficing

Technically, boundary logic does not generate exact minimization, instead it generates *satisficing solutions*, minimizations for which pragmatic criteria are reached, although effectively infinite effort could do better.

To understand how this is accomplished in detail requires an explanation and an understanding of a highly innovative but tractable boundary logic algorithm called *virtual insertion*. BL uses computational techniques which only a few people in the world have seen, and which are so fundamentally different from those in common academic and industrial usage that understanding them may require diligent study.

However, the claim of boundary logic is that it is inherently simpler. It is thus an obligation that, given the right perspective, frame of mind, and conceptual training, virtual insertion is ultimately very simple and very easy to understand.

The following three sections describe how boundary logic achieves what some may consider to be computational magic. They are written from a pedagogical, rather than from a mathematical, point of view. The first, *Boundary Logic*, includes basic perspectives, definitions, and some simple examples. Next, *Elusive Complexity* uses boundary logic tools and techniques to demonstrate that, seen from the perspective of boundary logic, the conventional rules of inference, the Boolean equivalence formulas, and the common proof strategies are all close to trivial. Finally, *Virtual Insertion* describes the central polynomial computational innovation of boundary logic, and then specifies exactly where the boundary between intractable and tractable computation occurs.

BOUNDARY LOGIC

This section presents the basic ideas of boundary logic, and provides several simple examples.

The Nature of Boundary Logic

Boundary logic (BL) is a calculus of containers (mathematically, partially ordered lattices). A container can be interpreted as logical *nor* with an arbitrary number of arguments. Although BL therefore can be *read* as conventional logic, to do so obscures the powerful transformations that arise only from a container-based interpretation. Formally, BL is more succinct than conventional logic, the mapping from logical forms to containers is many-to-one. The simplicity of the formal model translates directly into simplicity in data structures and algorithms.

BL is utterly simple, once a few perspective changes have been accepted. These are simply stated:

• Rather than *True* and *False* (or 1 and 0), BL uses *presence* and *absence*. Absence permits half of a binary system to simply not exist.

• To manipulate presence and absence, basic objects require an *inside* and an *outside*, thus containers. Conventional variables lack an inside.

• Textually, a parenthesis (called ***parens***) is a container. The following parens contains the objects A and B: (A B). And this parens contains nothing: ( ). When expressed in boundary notation, all logic expressions are simply well-balanced parens.

• Containers are spatial objects which *do not possess* ordering, grouping, or cardinality properties (commutativity, associativity, arity). The *only* property that characterizes containers is that of *containment*.

• Parens forms are transformed and simplified by *deletion* (erasure) operations rather than by rearrangement or by accumulating facts.

• Transformational rules rely on the *transparency* property of parens: everything inside a nest of parens containers is transparent and accessible to everything which is outside.

• There are only three BL transformations, all of which are both visually and algorithmically simple.

Nesting of parens is equivalent to multiple levels of logic, that is, to logic expressions embedded inside other logic expressions. In general, well-formed parens are trees. By enforcing unique objects without representational replicates, parens forms are directed acyclic graphs. Thus a parens form can be understood to be a succinct representation of a logic graph or network.

The Map from Logic to Parens

Logical *nor* is containment:

not (a or b or c)	(a b c)

From the single mapping of *nor* to parens, the parens form of the other logical connectives is easily deduced. The *nor* of a single argument is logical *not*:

not a	(a)

The *nor* of no arguments is logical *True*:

True ( )

Since a parens therefore negates its inside, the *True* empty parens is the negation of nothing. That is, logical *False* has no representation, it is absence. The absence of *True* is *False*:

False

Logical *or*, as the absence of the negation imposed by *nor*, is sharing space:

a or b or c a b c

In BL, forms are *disjoint* in that space provides no connective or relational structure. Sharing space is defined solely as being the contents of the same container. Forms sharing the same space can be processed in parallel. Logical *and* is formed by containing a space and every object within that space:

a and b and c ((a)(b)(c))

Finally implication, logical *implies*, is expressed simply as the distinction between inside and outside. The inside of a boundary implies the outside.

a implies b (a) b

Therefore any combinational logic form can be expressed as a well-formed parens form with variables. Table I presents the map from Boolean to boundary logic.

It is illustrative to transform the DeMorgan law into parens:

a and b = not ((not a) or (not b)) ((a)(b)) = ((a)(b))

That is, expressions which are structurally different in conventional logic may not differ in the simpler boundary logic.

To understand void-equivalence and the role of non-representation, the following transcription is useful:

(False or (False or False)) implies False ( )

Other examples of transcription between conventional logic and boundary logic follow:

a iff b (version 1)	(a b) ((a)(b))
a iff b (version 2)	((a (b))((a) b))
if a then b else c	(((a) b)(a c))
2/3majority(a,b,c)	((a b)(a c)(b c))
all-equal(a,b,c)	(a b c) ((a)(b)(c))

In transcribing logic expressions to parens, *square brackets* are used solely for pedagogical purposes, to make the transcription more readable by highlighting particular structures. Square brackets *are* parens.

BL is an algebraic system, function composition is achieved by substitution. Consider this transcription as an example. Square brackets highlight the structure of if-then-else:

if a then 2/3majority(b,c,d) else all-equal(b,c,d)

[[[a] ((b c)(b d)(c d))] [a (b c d) ((b)(c)(d))]]

Transformation of Parens Forms

All of the mechanism of logical inference, Boolean transformation, and proof strategy reduces to three BL deletion equations which can be considered to be the *axioms* of boundary logic:

(A ( ))	**=**	**<void>**	***OCCLUSION***	**occ**
((A))	**=**	**A**	***INVOLUTION***	**inv**
A {B A}	**=**	**A {B}**	***PERVASION***	**per**

Capital letters refer to any parens form regardless of complexity, including spatial collections of separate forms and the absence of any forms. Each equation proceeds from left to right via *deletion of structure*. The curly braces in Pervasion convey the *transparency rule*, they stand in place of any intervening content at the same or a deeper level of nesting. Pervasion is a

multilevel simplification rule: what is outside can be deleted or inserted inside, independent of the depth of nesting within a parens form. Some examples of Pervasion follow. The layout of these examples uses space to emphasize which parens structures have been deleted.

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

     a (b (c (a d)))
==>  a (b (c (  d)))

     (a b) (c (d (e (a b))))
==>  (a b) (c (d (e      )))
```

Table IIa presents the conventional axioms of inferential logic with their transcription to boundary logic. What appears to be a relatively complex axiomatization is shown to be compound applications of the three simple BL axioms. Table IIa also presents the axioms of Boolean algebra, a group-theoretic approach to logic. These are also transcribed into BL with similar results, all are derivable from the boundary axioms. Table IIb presents three separate but quite related axiomatic bases of boundary logic, including the one above.

For illustration, here is the BL approach to a Boolean minimization problem expressed in the symbolism of both logical connectives and Boolean algebra. Spacing is again used to emphasize deletions.

b or ((not a) and b) or ((not b) and c) or (c and d)

f(a,b,c,d) = b + a'b + b'c + cd

```
b (((a))(b)) (((b))(c)) ((c)(d))          transcribe
b (  a  (b)) (  b  (c)) ((c)(d))          inv
b (  a  ( )) (     (c)) ((c)(d))          per b
b            (     (c)) ((c)(d))          occ
b                   c   ((c)(d))          inv
b                   c   (( )(d))          per c
b                   c                     occ
```

f = b + c transcribe

b or c

All BL algorithms are of low degree polynomial complexity, basically sorting and pattern-matching. BL removes sufficient redundancy from logical expressions so that only rare pathological expressions require the extra non-polynomial effort. For example, boundary logic algorithms produce world-class minimization without exponential processes. Boundary logic is a simple, tractable multilevel calculus for Boolean minimization.

ELUSIVE COMPLEXITY

This section is an exploration which attempts to explicitly locate the boundary between *exponential logical complexity* and *polynomial logical simplicity*. The strategy is to identify those logical expressions that may harbor the need for exponential computational effort in the course of determining whether or not they are tautologies.

Apparently complex logical expressions can be transcribed into parens notation and reduced to their minimal form using polynomial BL algorithms. When the parens form of a logical expression requires exponential reduction effort have we reached the source of computational complexity. As it turns out, this occurs only for large and convoluted logical expressions, ones that are rarely encountered in pragmatic applications of logic such as deduction problems, database queries, expert systems, and circuit schematics.

Since BL is an algebraic system, parens forms are algebraically minimized during the tautology identification process. The BL reduction process is more powerful than simple tautology identification. The same polynomial computational effort used to untangle potential tautologies also removes irrelevant logical structure, leading to approximate minimal forms. The minimization is not exact, since in the BL process that would be analogous to Boolean factoring, it is common to fall into a local minimum which requires exponential effort to escape. However, these local minima are generally quite close to the global minima. Thus, *pragmatically*, Boolean minimization is tractable, requiring only simple and efficient algorithms.

The purpose of showing all the BL proofs in the following is to demonstrate that none require NP processes, in fact almost all require little thought beyond identifying variables which occur both outside and inside nested parens. The boundary logic proofs toward the end of this section look quite complex, however, by following the sequence of deletions visually, they become relatively easy to follow.

Irrelevant Variables

It is easy to see how variables can occur in a logical expression without contributing to the meaning of the expression. We begin with simple expressions and progress through all conventional logical theorems. Each

logic expression on the left is accompanied by the equivalent parens form on the right.

Dominance of Truth

(A or True) = True	A ( ) = ( )

Any logical expression joined by disjunction with *True* is a tautology. This type of irrelevance may be buried in multiple disjunctions:

(A or (B or (C or True))) = True	A B C ( ) = ( )

For both of the above logic expressions, the equivalent boundary theorem is Dominion:

A ( ) = ( )	***DOMINION***	**dom**

In parens notation, the capital letter A stands for any parens form, including a collection of forms or no forms at all.

Logical Addition

A -> (A or B) = True	(A) A B = ( )

The logical law of Addition permits the inclusion of arbitrary forms through disjunction, without implicating a logical ground. From the BL perspective, this is a simple application of Pervasion followed by Dominion.

A {B A} = A {B}	***PERVASION***	**per**

The curly braces of Pervasion represent any degree of parens nesting. Thus, by setting {B} to <void>, Pervasion becomes the logical law of Idempotency, called Replication in BL:

A A = A	***REPLICATION***	**rep**

We will use the default interpretation of parens forms, that if they exist they are intended to be *True*. That is, after reduction, existent parens forms are always satisfiable. Tautological forms reduce to the form of boundary logic truth, an empty parens: ( ). The empty parens is called a *mark*. Contradictory forms vanish into the boundary logic void. Forms that need not exist, that is which are equivalent to logical *False*, are called *void-equivalent* forms. The BL proof of Logical Addition follows:

(A) A B	transcribe
( ) A B	per A
( )	dom

Excluded Middle

(A or ~A) = True	A (A) = ( )

The Law of Excluded Middle is fundamental to conventional logic, however boundary logic does not share the same historical evolution. Excluded Middle is not an important BL theorem. Instead it is a simple proof, identical to that of Logical Addition shown above.

A (A)	transcribe
A ( )	per A
( )	dom

In conventional logic, the form of A, however, may look different in different locations, due to other logical equivalences. For example, consider the Symmetry-of-And:

A and B = B and A	((A)(B)) = ((B)(A))

Combining this with Excluded Middle generates a more complicated logical expression:

((A and B) or (not (B and A)) = True	((A)(B)) (((B)(A)))

This complexity is pragmatically removed by canonical ordering. Since sorting can be achieved efficiently, this type of logical embedding is not the source of computational complexity.

The typographical conventions of parens create an appearance of sequential ordering, however BL is formulated in featureless space. It does not matter which components of the form "come first", since BL does not possess an ordering concept. Pragmatically, when implementing BL in parens notation, all forms are first standardized into a canonical sequential order. The rules that define the canonical ordering are arbitrary, and can be customized to particular implementation environments and data structures. As an analogy, consider that the members of a set also have no ordering concept, although ordering is forced upon a set when it is represented as a list or as an array. Thus the reduction of this compound logical tautology is identical to those above:

```
((A)(B)) (((B)(A)))                transcribe
((A)(B)) (((A)(B)))                sort
((A)(B)) (         )               per ((A)(B))
         (         )               dom
```

We begin to see here what would be expected from an inherently simpler system: forms that are different in an inefficient representation are not different in an efficient representation; proofs that differ in a more complicated system do not differ in a simpler system such as boundary logic.

In the following, the artificial sort operation, which is solely an artifact of sequential implementation, will not be included in BL proofs.

More Ways to Hide Irrelevant Variables

BL is an algebraic calculus, its main connective is the equal sign, =. The algebraic form of a Boolean equivalence asserts that two expressions are equal no matter what values are assigned to the variables. Logical Equivalence converts any BL equation into a pure parens form.

Logical Equivalence

A = B iff (A -> B) & (B -> A) = True

A = B iff (((A) B)((B) A))

It is not necessary to assert the truth of such a parens form, since if it is *False*, it will vanish. By default, existent parens forms are assumed to be *True*.

In the Boolean domain, algebraic equivalences can always be converted into Truth statements in *standard form* by applying the Logical Equivalence transformation. This permits hiding irrelevant variables in yet another way. Using the Symmetry-of-And as an example:

((A and B) -> (B and A)) & ((B and A) -> (A and B)) = True

Here, square brackets emphasize the parens form of Logical Equivalence. The proof is straight-forward:

```
[ [[((A)(B))] ((B)(A))] [[((B)(A))] ((A)(B))] ]       transcribe
[ [[        ] ((B)(A))] [[        ] ((A)(B))] ]       per ((A)(B))
[                                               ]       occ
```

The final line of proof uses the Occlusion axiom:

(A ( )) = <void> ***OCCLUSION*** **occ**

For completeness, together with Pervasion and Occlusion, the third and only other BL axiom is Involution:

((A)) = A ***INVOLUTION*** **inv**

Absorption

(A and (A or B)) = A ((A)(A B)) = A

The logical rule of Absorption asserts a Boolean equivalence. Algebraic forms are converted into tautologies by the logical process of necessary and sufficient conditions; above this was identified as Logical Equivalence. Converting logical Absorption to standard form yields

((A and (A or B)) -> A) and (A -> (A and (A or B))) = True

The BL proof uses square brackets to highlight the form of Logical Equivalence:

```
[ [[(A) (A B)]] A] [[A] ((A) (A B))] ]          transcribe
[ [  (A) (A B)   A] [[A] ((A) (A B))] ]          inv
[ [  ( ) (  B)   A] [[A] ((A) (A B))] ]          per A
[                   [[A] ((A) (A B))] ]          occ
                     [A] ((A) (A B))             inv
                     [A] (    (A B))             per (A)
                     [A]       A B               inv
                     [ ]       A B               per A
                     [ ]                         dom
```

Although Absorption and Involution both stumped the theorem proving community up until the 1980s, parens notation shows them both to be simple on both sides of the necessary and sufficient conditions proof.

In general, to facilitate pattern-matching, BL transforms are expressed without outer bounds. Thus BL Absorption is stated as

(A) (A B) = (A) ***SHALLOW ABSORPTION***

Pervasion

It is the algebraic proof of Absorption (below) which first suggests non-trivial computation. The axioms of Occlusion and Involution are both inherently simple, it is the axiom of Pervasion which houses all notions of Boolean intractability.

A (A B) = A (B) ***SHALLOW PERVASION***

Pervasion can operate in two directions: by erasing forms that are deeper than (pervaded by) identical forms, and by inserting replicates of forms existent in an exterior space into any interior space. Since erasure cannot contribute to complexity, it is the insertion operation which is potentially costly.

Consider the left-side of Absorption. How can it be directly transformed into the right-side? No simple BL rules appear to apply. To move forward with an algebraic proof, we must insert the exterior (A) into the interior space. We will call the constructive use of Pervasion *insertion* (labeled *ins* in proofs). An inserted form is highlighted by carets, ^...^.

(A) (A B)	left-side
(A) (^(A)^ A B)	ins (A)
(A) (^( )^ A B)	per A
(A)	occ, right-side

An important understanding is that insertion is *virtual* rather than literal. That is, the inserted form is never actually transcribed into the inner form. Instead, it is inserted hypothetically, to examine the consequences. As the virtual form reduces through the effect of pervading variables in its environment, it may either

1) vanish, in which case it has no reducing effect

2) reduce to a mark, ( ), in which case the *context* of the virtual mark is erased by Occlusion, or

3) reduce to a non-ground form, one still containing variables, in which case it has no direct reducing effect.

How a virtual form reduces cannot be predicted in advance. The inserted form may contain more variables than the form being inserted into, these extra variables may be eliminated in the course of insertion reduction. The inserted form may contain fewer variables than the form being inserted into, the reduction of the inserted form to a mark may occur at a shallower depth than some of the unmentioned variables. Thus, the effect of insertion is

empirical, but at no time is it non-polynomial since insertion is simple tree descent without backtracking.

Instead of a virtual insertion proof, the effect of bounded forms on the forms they share space with can be expressed as a theorem. The BL Absorption theorem was visited in a less general form above.

(A) {C (A B)} = (A) {C} ***ABSORPTION*** **abs**

Absorption, like Pervasion, is a deep theorem, relying on the transparency of parens to their contexts. The capital letters A,B, and C each represent arbitrary parens forms.

In the case of natural deduction, adding theorems leads to an explosion in computation effort, since which theorem to apply at which point must be identified, and this may result in a trial and error process which is the hallmark of NP computation. In BL, Absorption is implemented by the Pervasion algorithm; from a BL perspective they are actually the same rule. In any event, all BL theorems introduced herein can be implemented using low degree polynomial algorithms for sorting, pattern-matching, and substitution. For the purpose of identifying the source of NP complexity, the BL theorems are all polynomial, as is any compound combination of them.

Deep Transformations

Like Pervasion and its sister Absorption, most boundary logic theorems are *deep*, in that they are insensitive to depth of nesting. The two exceptions are Distribution and Pivot, which are discussed in later sections.

Deep transformations greatly simplify proof and minimization by rendering intervening parens transparent while not increasing computational effort. Deep rules are extremely powerful and have no parallels in conventional logical theorems. Consider deep Absorption directly above. The curly braces indicate that {C} can be any arbitrary intervening content or depth of nesting. Here are several examples of increasing apparent complexity. Forms deleted via Absorption are highlighted by spaces:

```
     (a) (c (d (e (a b)))
==>  (a) (c (d (e      ))

     (a b) (c (d (e (a b f g))))
==>  (a b) (c (d (e          )))

     (a b) (c (a d (e (b f))))
==>  (a b) (c (a d (e      )))
```

The last example illustrates another feature of deep rules: *constituent forms do not need to share the same space*. We can refine the definition of Absorption to emphasize this:

(A B) {C A {D (B E)}} = (A B) {C A {D}} ***ABSORPTION***

In the above equation, the {C} form indicates that the location of the embedded A form can be arbitrarily deep. The {D} form indicates that the location of the embedded B form can be arbitrarily deeper again than the A form. That is, the deleted (B E) form does not have to be in the same space as the embedded A. This is substantially different than all conventional equational or logical transformations, which express rules for expressions only at the same level of depth.

Here, the typographical notation begins to show signs of weakness. Although we will continue to use textual forms herein, it may be more convenient to express BL forms and transforms as graph reduction algorithms.

A deep rule can be proved as a theorem by successive application of shallow rules. Consider a proof of deep Absorption, expressed without depth braces:

```
(A B) (        C (        A (      D (      B E))))
(A B) (^(A B)^ C (        A (      D (      B E))))        ins (A B)
(A B) (^(A B)^ C (^(A B)^ A (      D (      B E))))        ins (A B)
(A B) (^(A B)^ C (^(  B)^ A (      D (      B E))))        per A
(A B) (^(A B)^ C (^(  B)^ A (^(B)^ D (      B E))))        ins (B)
(A B) (^(A B)^ C (^(  B)^ A (^(B)^ D (^(B)^ B E))))        ins (B)
(A B) (^(A B)^ C (^(  B)^ A (^(B)^ D (^( )^ B E))))        per B
(A B) (^(A B)^ C (^(  B)^ A (^(B)^ D           )))        occ
(A B) (^(A B)^ C (^(  B)^ A (      D           )))        per (B)
(A B) (^(A B)^ C (^(A B)^ A (      D           )))        ins A
(A B) (^(A B)^ C (        A (      D           )))        per (A B)
(A B) (        C (        A (      D           )))        per (A B)
```

This approach is a fundamental miscomprehension of boundary logic theory in two distinct ways. First, the careted form ^(A B)^ is virtual, it is void-equivalent and does not exist in the form. Thus it need not be Occluded to be erased. Virtual forms are simply abandoned into <void> after they have served their reduction purpose. This point is illustrated by changing the steps of the above proof which follow Occlusion:

```
(A B) (^(A B)^ C (^(  B)^ A (^(B)^ D           )))        occ
(A B) (        C (        A (      D           )))        virtual
```

Second, deep rules are independent of depth of nesting. Parens are *transparent* to the outside, they are computationally semipermeable. The intermediate copies of (A B) should not be recorded, since only one replicate need exist for insertion. Thus, the appropriate proof of deep Absorption calls upon deep Pervasion directly:

```
(A B) (       C (       A (     D (       B E))))
(A B) (       C (       A (     D (^(A B)^ B E))))      ins (deep)
(A B) (       C (       A (     D (^(  B)^ B E))))      per A (deep)
(A B) (       C (       A (     D (^(   )^ B E))))      per B(shallow)
(A B) (       C (       A (     D             )))      occ
```

This understanding permits the curly braces generalization to Absorption, where curly braces represent any number of levels of intervening parens, and the label attached to the curly braces represents any arbitrary form(s):

```
(A B) {C        (       A {D      (       B E)})}
(A B) {C        (       A {D                  })}      abs
```

A subtle question is whether or not it is possible to tangle forms of Absorption recursively to generate an intractable tautology. The computational implications of transparency assure that this cannot be the case, since tangled forms will continue to be pervaded by forms which reduce them.

The Simplicity of Logical Deduction

Moving forward, we enter the territory of logical inference rules, replacement rules, and proof strategies. First we consider a compound form of Excluded Middle, sometimes called the Law of Cases:

Compound Excluded Middle (Cases)

((A or B) and (A or (not B))) = A ((A B)(A (B))) = A

A BL rule of shallow Compound Excluded Middle might be stated as

(A B) (A (B)) = (A) ***SHALLOW CASES***

In BL however, this is not a theorem, it is simply an instance of the Absorption rule.

The BL proof of Shallow Cases using standard form follows:

```
[ [[((A B)(A (B)))] A] [[A] ((A B)(A (B)))] ]        transcribe
[ [  (A B)(A (B))   A] [[A] ((A B)(A (B)))] ]        inv
[ [  (  B)(  (B))   A] [[A] ((A B)(A (B)))] ]        per A
[ [  (  B)(     )   A  [[A] ((A B)(A (B)))] ]        per (B)
[                      [[A] ((A B)(A (B)))] ]        occ
                        [A] ((A B)(A (B)))           inv
                        [A] (           )            abs (twice)
                            (           )            occ
```

All the Inferential Rules

The tautologies of Boolean expressions are well catalogued, and implicitly we have already visited most of them. Table III presents the traditional inferential rules of logic, and their transcription into parens notation. Each parens form is reduced, and each reduces to a mark. (The final trivial application of Dominion has been omitted.) This table demonstrates that the rules of inference can be derived from repeated applications of the three BL axioms.

One significant difference between conventional inferential logic and BL is that conventional logic has a strong differentiation between syntax (what an expression looks like) and semantics (what an expression means). The syntax/semantics gulf is an artifact of symbolic representations which permit the form of an expression to be essentially arbitrary. Much of the development of modern logic theory has focused on the compatibility and computability of meaning and representation. BL could adopt a similar approach, however diagrammatic formalisms have another route: *the shape of a parens form is a picture of its meaning*. When interpreted as containers rather than as logic, parens forms are what they portray, which is a map of the spatial relationships between collections and nestings of bounded spaces. From the perspective of BM, logical inference, logical intent (symbolized by the single and double turnstiles, |= and =||=) and logical form are the same. BL transcriptions do not distinguish between types of distinctions, those used to convey valid inferential steps, and those used to convey the form of logical connectives. The transcriptions in Table III use a bold parens, **(...)**, to identify the container which represents the semantic inference. The BL proofs immediately ignore these bold forms, treating them as normal parens. For inferential purity, every parens proof can be formulated so that the final dominating parens is indeed the bold, semantic one.

Table IV transcribes and proves the replacement rules of conventional logic. In Table III, proof is by reduction to a mark. The bidirectional inference of replacement rules converts to the BL equal sign. Proof in Table IV is by reduction of an equation to an identity. Proofs in Tables III and IV are

intended to illustrate the simplicity and the similarity of rules which have evolved as separate principles in conventional logic. All are nearly trivial combinations of the three simple BL axioms.

A few logical replacement rules have non-trivial algebraic proofs. Some of these are included in Table V. The strategy in most of these proofs is conversion of one side of an equation to the other. Expressing any of these proofs in standard form reduces the difficulty of the proof substantively.

Distribution

((A or B) and (A or C)) = (A or (B and C)) ((A B)(A C)) = A ((B)(C))

One commonly occurring transformation is the rule of Distribution. Table V presents an algebraic proof of Distribution, the standard form proof follows. The structure of this rule immediately suggests potential new complexities. There are three variable forms rather than two, the right-side is not inherently simpler than the left-side (although it does contain one less reference), and the rule itself suggests an irreducible entanglement. We examine its standard form proof structure:

```
[ [[((A B)(A C))] A ((B)(C))] [[A ((B)(C))] ((A B)(A C))] ]     transcribe
[ [  (A B)(A C)   A ((B)(C))] [ A ((B)(C))] ((A B)(A C))] ]     inv
[ [  (  B)(  C)   A ((B)(C))] [[A ((B)(C))] ((A B)(A C))] ]     per A
[ [  (  B)(  C)   A (      )] [[A ((B)(C))] ((A B)(A C))] ]     per (B)(C)
[                             [[A ((B)(C))] ((A B)(A C))] ]     occ
                               [A ((B)(C))] ((A B)(A C))        inv
```

One direction of the logical equivalence has shown to be trivial to prove. To make progress in proving the other direction we must again insert an outer form into an inner form:

```
(A ((B)(C))) ((               A B)(A C))                    subproblem
(A ((B)(C))) ((^(A ((B)(C)))^ A B)(A C))                    ins
(A ((B)(C))) ((^(  (( )(C)))^ A B)(A C))                    per A B
(A ((B)(C))) ((^(          )^ A B)(A C))                    occ
(A ((B)(C))) (                   (A C))                     occ
(A ((B)(C)))                      A C                       inv
(  ((B)( )))                      A C                       per A C
(          )                      A C                       occ
(          )                                                dom
```

We restate the BL rule of Distribution:

A ((B)(C)) = ((A B)(A C)) ***DISTRIBUTION*** **dis**

From the BL perspective, separating forms is indeed a simplification. Thus, BL Distribution has a preferred direction, here from right to left. In logic, Distribution is an axiom, it is considered to be fundamental. The BL proof decomposes it into a simpler form by showing that it can be derived from the three BL axioms. Here too, BL has exposed an unnecessary complication which has been built into the foundation of conventional logic.

Looking Deeply for Complexity

The proof of Distribution is non-trivial, but not complex. Boolean intractability may perhaps require a subtle mixture of Distribution and other, perhaps complicated, logical theorems. This observation would be esoteric, but for the fact that the combination of Distribution and Cases is a primary construct in programming languages as the *If-then-else* statement, and in circuitry as the *Mux* gate.

(if A then B else C) =def= ((A -> B) and ((not A) -> C)) (((A) B)(A C))

In the boundary form of If-then-else we can see the shape of Cases,

```
((A B)(A (B)))          Cases
(((C A)(B (A)))         If-then-else
```

Parens forms provide a capability not available to conventional logic: the location of parens can be compared for the same *shape*, while ignoring the particular variable names.

An implication of If-then-else derives from the Excluded Middle; since the expression A is in both polarities, either B or C must occur:

(if A then B else C) -> (B or C)

The BL proof of this implication uses square brackets to highlight the form of implication:

```
[(((A) B)(A C))] B C                    transcribe
  ((A) B)(A C)   B C                    inv
  ((A)  )(A  )   B C                    per B C
  (     )(A  )   B C                    per (A)
  (     )                               dom
```

In BL, the If-then-else implication is expressed in an even more general form, called Consensus:

Consensus

(A B) (C (A)) (B C) = (A B) (C (A)) ***SHALLOW CONSENSUS***

The standard form proof is straight-forward (a smaller font is now desirable to contain the proof display to single lines):

```
[ [[(A B) (C (A)) (B C)] (A B) (C (A))] [[(A B) (C (A))] (A B) (C (A)) (B C)] ]    trans
[ [[             (B C)] (A B) (C (A))] [[            ] (A B) (C (A)) (B C)] ]    per
[ [               B C   (A B) (C (A))]                                       ]    occ
                  B C   (  B) (  (A))                                             inv
                  B C   (  B) (     )                                             per
                              (     )                                             dom
```

The form of deep Consensus emphasizes that neither of the consensual forms (A B) or (C (A)), nor the implied forms B and C need share the same space. All that is necessary is that the deepest form be one of the implied forms.

(A B) {D B {E (C (A)) {F (C G)}}} = (A B) {D B {E (C (A)) {F}}}

CONSENSUS **con**

Should both consensual forms occur in shallower spaces than both implied forms, then both implied forms are void-equivalent, as well as any other forms sharing their space. The algebraic proof of deep Consensus is instructive.

```
(A B) {D B {E (C (A)) {F (               C G)}}}
(A B) {D B {E (C (A)) {F (^(A B)(C (A))^ C G)}}}          ins
(A B) {D B {E (C (A)) {F (^(A  )(  (A))^ C G)}}}          per B C
(A B) {D B {E (C (A)) {F (^(A  )(     )^ C G)}}}          per (A)
(A B) {D B {E (C (A)) {F                   }}}          occ
```

Technically, Consensus as well is not a separate theorem in BL, it is simply the accumulation of absorbed forms through deep nesting.

The two consensual forms, (A B) and (C (A)) also have the shape of Distribution, but with the distributed form A in two polarities, rather like Excluded Middle. This creates a generalization of Distribution, called Biconditional Negation in logic. It is called Pivot in BL terms. The algebraic proof is presented in Table V, the standard form proof follows.

Pivot

((A or B) and (C or (not A))) = (((not A) and B) or (A and C))

((A B)(C (A))) = (A (B)) ((A)(C)) ***PIVOT*** **piv**

The proof using standard form:

```
[ [[((A B)(C (A)))] (A (B)) ((A)(C))] [[(A (B))((A)(C))] ((A B)(C (A)))] ]
[ [  (A B)(C (A))   (A (B)) ((A)(C))] [[(A (B))((A)(C))] ((A B)(C (A)))] ]       inv
```

Here the simple reduction appears to stop. A closer look yields an application of deep Absorption in the first subform:

```
  [  (A B) (C (A))  (A (B)) ((A)(C))]                                  subproblem
  [  (A  ) (C (A))          ((A)(C))]                                  abs
  [  (A  ) (C    )          (   (C))]                                  per (A)
  [                         (     )]                                   per (C)
              <void>                                                   inv
```

This subform vanishes, leaving the following:

```
[                                [[(A (B))((A)(C))] ((A B)((A) C))] ]        prob
                                  [(A (B))((A)(C))] ((A B)((A) C))           inv
```

Again we insert the outer form and reduce:

```
                  ((A (B))((A)(C))) ((^((A (B)) ((A)(C)))^ A B) ((A) C))          ins
                  ((A (B))((A)(C))) ((^((  ( )) (( )(C)))^ A B) ((A) C))          per A B
                  ((A (B))((A)(C))) ((^(               )^ A B) ((A) C))          occ
                  ((A (B))((A)(C))) (                         ((A) C))          occ
                  ((A (B))((A)(C)))                            (A) C            inv
                  ((A (B))(   ( )))                            (A) C            per (A) C
                  ((A (B))       )                             (A) C            inv
                    A (B)                                      (A) C            inv
                    A (B)                                      ( ) C            per A
                                                               ( )              dom
```

In general, all recursive embeddings of logical theorems in themselves and in each other reduce with polynomial effort, as do any deep nestings of subforms involved in the theorems. The key is the transparency of parens.

Tangles

There are two *boundary logic tangles*:

(A B) ((A)(B)) = ((A (B))((A) B)) ***PIVOT***

A ((B)(C)) = ((A B)(A C)) ***DISTRIBUTION***

Tangles cannot have a deep counterpart, since deep transformations rely upon the transparency of parens, and tangling is simply another way of recognizing when transparency is not applicable. The NP complexity of tautology identification does not stem from these tangles when, in the course of a proof, they are applied in one direction only. BL has a preferred direction, the one in which a single form is partitioned, or untangled, into two forms, right to left in both rules above.

A significant problem with these rules is that the capital letters are arbitrarily complex forms themselves, which are required to be equivalent in value only, not in structure. Therefore, a structural approach such as pattern-matching could fail. For example, Distribution can be recursively tangled within itself:

X ((Y)(Z)) = ((X Y) (X' Z))

X = ((A B) (A C))
X' = A ((B)(C))

((A B) (A C)) ((Y)(Z)) = ((**((A B) (A C))** Y) (**A ((B)(C))** Z))

Although such tangles may add significantly to computational effort, Pervasion, and the transparency of parens, is sufficient to untangle them, as was demonstrated in the previous section. Embedded instances of both Distribution and Pivot untangle because both theorems are based on Pervasion, itself a deep rule.

How Can Irrelevant Variables Hide in Intractable Ways?

Consider the recursive embedding of If-then-else forms, that is, Distribution of If-then-else.

Distribution of If-then-else

(if (if A B C) D E) = (if A (if B D E) (if C D E))

Transcription is in parts for readability:

```
(if X D E) = (if A Y Z)

             X = (if A B C) = ((A C) ((A) B))
             Y = (if B D E) = ((B E) ((B) D))
             Z = (if C D E) = ((C E) ((C) D))

(if X D E) = ((      X         E) ((      X        ) D))
           = ((((A C) ((A) B)) E) ((((A C) ((A) B))) D))          subst
           = ((((A C) ((A) B)) E) (  (A C) ((A) B)   D))          inv

(if A Y Z) = ((A       Z        ) ((A)       Y        ))
           = ((A ((C E) ((C) D))) ((A) ((B E) ((B) D))))          subst
```

The entire problem stated as a parens equation is therefore:

```
((((A C)((A) B)) E) ((((A C)((A) B))) D)) = ((A ((C E)((C) D))) ((A) ((B E)((B) D))))
```

The proof as well is in parts, -> and <-, using square brackets to highlight the implication in each direction:

Proof ->:

```
[((((A C)((A) B)) E) ((A C)((A) B) D))] ((A ((C E)((C) D))) ((A)((B E)((B) D))))
  (((A C)((A) B)) E) ((A C)((A) B) D)   ((A ((C E)((C) D))) ((A)((B E)((B) D))))    inv
```

There are three separate forms, two formerly inside the Involution, one outside. Here a new type of complexity might arise: which forms should be inserted into which? This issue is addressed in depth in the following section. The answer is *set-insertion*, insert *all* exterior forms into the interior of all others. This may add significant overhead, but does not yet step into exponential effort. Since all inserted forms are virtual, should they fail to trigger a reduction, they can simply be discarded into the void.

Insert both the first and the second subforms into the third. Note that the third subform is itself composed of two components. Below the insertion visits each of them separately. The inserted pair still exists and is saved to the side.

```
((^(((A C)((A) B)) E) ((A C)((A) B) D)^ A ((C E)((C) D))) ((A)((B E)((B) D))))    ins
((^(((  C)(( ) B)) E) ((  C)(( ) B) D)^ A ((C E)((C) D))) ((A)((B E)((B) D))))    per A
((^(((  C)       ) E) ((  C)       D)^ A ((C E)((C) D))) ((A)((B E)((B) D))))    occ
((^(    C          E) ((  C)       D)^ A ((C E)((C) D))) ((A)((B E)((B) D))))    inv
((^(    C          E) ((  C)       D)^ A (           )) ((A)((B E)((B) D))))    per
(                                                        ((A)((B E)((B) D))))    occ
```

When inserted into, the first component of the third subform vanishes, so we insert again into the remaining component:

```
(                 (^(((A C)((A) B)) E) ((A C)((A) B) D)^ (A)((B E)((B) D))))    ins
(                 (^(((A C)(    B)) E) ((A C)(    B) D)^ (A)((B E)((B) D))))    per (A)
(                 (^((     (    B)) E) (     (    B) D)^ (A)((B E)((B) D))))    abs
(                 (^(           B   E) (     (    B) D)^ (A)((B E)((B) D))))    inv
(                 (^(           B   E) (     (    B) D)^ (A)(           )))    per
(                                                                      )    dom
```

Rather than vanishing, the second insertion leaves a mark. When this is recombined with the original two inserted forms which still exist, the entire expression reduces:

```
            (((A C)((A) B)) E) ((A C)((A) B) D)   ( )
                                                  ( )                          dom
```

We have demonstrated that the proof -> is indeed *True*. Before going on to the proof <-, observe that both component insertions were necessary to reduce the expression. Although this appears to double the effort, the increase is only linear in the number of forms to be inserted into, and thus still polynomial. Strictly, this effort was necessary due to the size of the original problem, rather than due to the insertion algorithm. Continuing,

Proof <-:

```
      [((A ((C E)((C) D))) ((A)((B E)((B) D))))] ((((A C)((A) B)) E) ((A C)((A) B) D))
        (A ((C E)((C) D))) ((A)((B E)((B) D)))   ((((A C)((A) B)) E) ((A C)((A) B) D))  inv
```

Again the first two subforms are set-inserted into the third, remembering that the first two subforms remain in existence:

```
   (((( ^(A ((C E)((C) D))) ((A)((B E)((B) D)))^ A C)((A) B)) E) ((A C)((A) B) D))     ins
   (((( ^(  ((   )(( ) D))) (( )((B  )((B) D)))^ A C)((A) B)) E) ((A C)((A) B) D))     per A C E
   (((( ^(  ((   )      ))                    ^ A C)((A) B)) E) ((A C)((A) B) D))     occ
   (((( ^(               )                    ^ A C)((A) B)) E) ((A C)((A) B) D))     occ
   (((                                             ((A) B)) E) ((A C)((A) B) D))     occ
   ((                                                (A) B   E) ((A C)((A) B) D))     inv
```

The inner component (A C) which received the insertion vanishes. Set-inserting now into the newly created component:

```
((   ^(A ((C E)((C) D))) ((A)((B E)((B) D)))^      (A) B   E) ((A C)((A) B) D))   ins
((   ^(A ((C E)((C) D))) (   ((   )(( ) D)))^     (A) B   E) ((A C)((A) B) D))   per
((   ^(A ((C E)((C) D))) (                )^      (A) B   E) ((A C)((A) B) D))   occ
(                                                          ((A C)((A) B) D))   occ
                                                            (A C)((A) B) D     inv
```

Let us collect what remains to examine the current state of the problem. We have the remnants of the third subform (above) and the two subforms which were the source of the virtual insertion:

```
(A ((C E)((C) D))) ((A)((B E)((B) D))) (A C) ((A) B) D
```

Continued set-insertion of the first two subforms into the smaller remaining remnants would fail. However, these newly created remnants have an immediate Pervasive impact on the two original subforms:

```
(A ((C E)((C) D))) ((A)((B E)((B) D))) (A C) ((A) B) D
(A ((C E)((C)  ))) ((A)((B E)((B)  ))) (A C) ((A) B) D                  per D
(A ((C E)  C    )) ((A)((B E)  B    )) (A C) ((A) B) D                  inv
(A ((  E)  C    )) ((A)((  E)  B    )) (A C) ((A) B) D                  per B C
```

There has been a surprising turn of events. The initial insertions did reduce the problem, but not completely. However, the remnants which resulted from the original insertions can now themselves be inserted back into the subforms which served as the initial set-insertions:

```
(A ((E) C) ^(A C) ((A) B)^) ((A) ((E) B)) (A C) ((A) B) D               ins
(A ((E) C) ^(  C) (( ) B)^) ((A) ((E) B)) (A C) ((A) B) D               per
(A ((E) C) ^(  C)        ^) ((A) ((E) B)) (A C) ((A) B) D               occ
(A         ^(  C)        ^) ((A) ((E) B)) (A C) ((A) B) D               abs
(A                        ) ((A) ((E) B)) (A C) ((A) B) D               virtual
(A                        ) (    ((E) B)) (A C) (    B) D               per (A)
(A                        )       (E) B   (A C) (    B) D               inv
(A                        )       (E) B   (A C) (     ) D               per B
                                                 (     )                dom
```

Virtual set-insertion was used maximally to reduce the problem form. This did not however result in failure, since the partial reductions created actual conditions sufficient to complete the proof. Even though the choice of which subform to insert has changed, the proof proceeds directly to smaller and smaller remnants. Computational complexity means that a problem gets more complex, not simpler. Thus, the above technique does not require exponential effort.

Finally, the following problem, Transitivity of Equality, illustrates an implicational proof.

Transitivity of Equality

((A = B) and (B = C)) -> (A = C)

```
[(((A B) ((A)(B))) ((B C) ((B)(C))))] (A C) ((A)(C))
  ((A B) ((A)(B))) ((B C) ((B)(C)))   (A C) ((A)(C))      inv
(A C) ((A)(C)) ((A B) ((A)(B))) ((B C) ((B)(C)))         rearrange
```

Note that there are four subforms in the transcription above. Inserting any one of them into the others achieves no reduction. Set-inserting all of them into either of the first two fails as well. Only when all are set-inserting into one of the two larger subforms does the proof succeed. This is because the reduction dependency is spread across all of the subforms, and is "threaded" only through the final two subforms. We cannot determine in advance which specific form will reduce via set-insertion. Fortunately, inserting all others into each subform does not yet involve exponential effort.

For a successful proof, we set-insert the first three subforms into the final one, and reduce:

```
   (A C) ((A)(C)) ((A B)((A)(B)))                          the inserted subforms

((^(A C) ((A)(C)) ((A B)((A)(B)))^ B C) ((B)(C)))                        ins
((^(A  ) ((A)( )) ((A  )((A)( )))^ B C) ((B)(C)))                        per B C
((^(A  )          ((A  )       )^ B C) ((B)(C)))                        occ
((^(A  )          (            )^ B C) ((B)(C)))                        per (A)
(                                  ((B)(C)))                           occ
                                    (B)(C)                               inv
```

Recombining with the inserted subforms:

```
(A C) ((A)(C)) ((A B) ((A)(B))) (B)(C)                                   reduced
(A C) ((A)   ) ((A B) ((A)   )) (B)(C)                                   per
(A C)   A      ((A B)   A     ) (B)(C)                                   inv
(  C)   A      ((  B)        ) (B)(C)                                   per A
(  C)   A      (             ) (B)(C)                                   per (B)
               (             )                                          dom
```

Set Insertion

For set-insertion to be completely general, *all* exterior subforms must be inserted into all interior forms. A form cannot be inserted into itself. During insertion reduction, care must be taken not to prematurely delete information which may assist the reduction. This restriction can be stated as a simple rule: during insertion reduction, apply Pervasion only one level

at a time. Do not use Absorption. As a subform descends into a nested interior space, Absorption will turn into Pervasion, so reduction is not weakened. This process traces the proof of shallow Absorption discussed earlier. Compare the first steps of the proof of Transitivity of Equality above, in which Pervasion is applied at all levels, to the following one-level Pervasion proof:

```
((^(A C) ((A)(C)) ((A B)((A)(B)))^ B C) ((B)(C)))                    ins
((^(A  ) ((A)(C)) ((A B)((A)(B)))^ B C) ((B)(C)))                    per C
((^(A  ) (   (C)) ((A B)(   (B)))^ B C) ((B)(C)))                    per (A)
((^(A  )     C    ((A B)     B  )^ B C) ((B)(C)))                    inv
((^(A  )     C    ((A  )        )^ B C) ((B)(C)))                    per B
((^(A  )     C    (             )^ B C) ((B)(C)))                    per (A)
(                                    ((B)(C)))                       occ
                                      (B)(C)                         inv
```

Although these constraints have not been used in this section, they certainly become important when addressing intractable tautologies in the following section.

Algorithm Discussion

Can the convoluted interplay of virtual insertions become exponentially complex?
Analysis of virtual set-insertion, including those insertions that result in wasted effort, yields an interesting result: although potentially complicated, the algorithm remains polynomial.

One method to generate NP complexity is to force the direction of either of the non-deep rules, Distribution and Pivot, *in both directions*. Then, from an algorithmic point of view, we would not know at any given step which direction to apply first. We would be forced to guess, and this would mean that we have entered the realm of non-polynomial search. However, these BL theorems, as well as all the others, have not been necessary for proof. None of the proof steps used thus far have called upon any of the more complicated theorems. The only mechanisms that are needed to get us to the current level of apparent logical complexity are the trivial Occlusion and Involution axioms and the Pervasion/Absorption axiom. Distribution/Pivot is not the key to complexity. Virtual set-insertion is sufficient alone.

Using the three axioms in a straight-forward manner as reduction rules requires polynomial effort. We would encounter exponential effort only if a problem required an exponential number of applications of these rules. Nothing in the parens forms thus far, no matter how complicated, requires this.

Virtual insertion has suggested in itself some potential sources of complexity. These are the issues:

- Failure of an insertion to reduce
- Discovered new insertions when a subform reduces but does not vanish
- Insertion into many different forms at many different depths
- Selecting which forms to insert into which
- Set-insertion, inserting all exterior forms into each interior form

First, consider a relatively complicated case, in which we have a large parens form. Inserting an exterior subform into all the spaces pervaded by that subform makes the large parens perhaps much larger, but this only multiplies the number of times the simple axioms must be applied. No parens form in itself requires an exponential number of insertions; in all cases the number of parens that express the problem is the upper limit on the number of required insertions. Clearly, a nested parens form does not grow exponentially when more variables are added (although flattened parens forms without nesting, which correspond to two-level logic expressions, do grow exponentially). When a subform reduces to remnants, the set-insertion algorithm may be reinitialized for the smaller problem, but since the smaller problem has fewer parens, the new effort is less, disqualifying the method as growing with exponentially complexity. Thus the first three issues do not introduce NP complexity.

The source of potential complexity may be in the selection process itself. Here is how that might happen: imagine that there are two large parens forms and that we must choose which to insert into which. One insertion may be a wrong guess while one may be a correct guess. Having to guess and then backtrack is one hallmark characteristic of NP problems. However this too is a false concern, *all* possible set-insertions can be made as a first step, even here the problem size has only been multiplied by some factor which relates to the number of parens in the problem statement. The key is to see that virtual insertions never enter themselves, and thus never generate an exponential growth. Set-insertion subsumes the selection question; it still is at most a multiplier of effort that does not introduce intractability.

One easy way to see that none of the above issues introduce complexity is to consider the parens form to be a directed acyclic graph. Parens are nodes, nestings are links. In this representation, virtual insertion is simply increasing the number of links in the graph, making it densely rather than sparsely connected. Reduction involves traversing the graph from top to bottom. Thus the graph algorithms which are analogous to the three BL axioms only traverse more links, they do not loop or backtrack. All that the addition of more links will cause is the linear increase of effort associated with doing the same thing more times. The key here is to see that every transformation deletes parens structure; deletion can never increase complexity.

We have yet to identify a logical tautology which requires exponential effort to reduce. BL heuristics still exist. It could be the case, for example, that always inserting the smaller forms into the largest form produces a reduction. Or perhaps always inserting the premise form into the conclusion form in a directional proof (-> or <-) will yield success. Or perhaps there is some method of counting the occurrences of specific variables which would always identify what to insert. The main point, however, is that these heuristics make a polynomial algorithm more efficient. They are not relevant to the search for exponential complexity. Moreover, we are not seeking *heuristics* here, for heuristics may fail to provide any reduction.

In order to finally find logical complexity, we will need to identify a problem for which virtual set-insertion does not succeed. This is the topic of the final section.

The current status can be summarized as follows. Virtual set-insertion is not *complete* in that it does not identify some tautologies. Those tautologies that it does identify include all combinations of all conventional logic theorems, complicated to any degree of mutual embedding.

We should expect that the construction of a non-reducible tautology is itself non-trivial. From research that has accumulated over the last thirty years, we know that many problems map onto (are the same as) the tautology problem, hence they are NP-complete. Large collections of these problems have been published. Many have particular names, such as the Pigeon-hole Problem, the Three-coloring Problem, the Traveling Salesman Problem, the Hamiltonian Cycle Problem, and the Satisfiability Problem.

The thesis of the next section is that it takes a NP-problem description, itself a difficult task, to cause virtual set-insertion to fail. This is very much a pragmatic, rather than a theoretical, result. Indeed, set-insertion does not help for the really hard problems, those known to be complex. However, it does distinguish between really hard problems and those that appear to be complex due merely to logical tangles.

This result has significant practical value. Experience has shown that practical problems, those constructed during the course of building practical applications and products, are almost always of the logical tangle type, and very rarely of the NP-complex type. Databases, expert system knowledge bases, circuit designs, designing patterns for clothing, seasonal scheduling of football games, arranging meeting times for busy executives, and a host of other pragmatic tasks all regularly run into logical tangles, but these tangles are rarely sufficient to create the need for an unavoidable exponential effort.

The benefits of the set-insertion algorithm run in two directions. Should the algorithm fail to reduce a problem, it is then known that the reduction

is not tractable. Should the algorithm succeed, then a problem for which other techniques rapidly become intractable has been solved efficiently.

For the purposes of this exploration, we have tacitly accepted that the technique of virtual set-insertion will indeed identify any tautology. The question has been solely whether or not the identification process is NP. In the following section, we will see how to make virtual set-insertion complete, so that it can solve even NP-complete problems. As might be expected, the generalization which makes set-insertion complete also makes it exponential.

VIRTUAL INSERTION

The previous section introduced a complicated example, Distribution of If-then-else, of virtual set-insertion. This final section further describes and illustrates this boundary logic proof technique. We identify two examples of tautological intractability, and provide this answer to the original question: the boundary between polynomial simplicity and exponential complexity is when the set-insertion algorithm must be applied recursively to insertions during the course of tautology identification. In closing, we briefly examine the role of symmetry in intractable problems.

A New Proof Technique

The virtual set-insertion algorithm as thus far presented is *not* a decision procedure for propositional logic like the other three widely known techniques (truth tables, natural deduction, and resolution). A decision procedure will always identify a tautology, whereas set-insertion fails for some tautologies, it is *not complete*. A *recursive generalization* of set-insertion is a complete decision procedure; however, it is also intractably complex. We will examine this later in the section.

Unlike the other known proof techniques, the algorithmic effort of set-insertion is polynomially bounded. Application of set-insertion discriminates between P and NP problems, it identifies logical tangles without relying on exponential processes. The conventional decision procedures, in an attempt to be complete, are all NP algorithms. In the case of truth tables, each variable evaluation step is exponential. For natural deduction, the selection of which rules to apply when is an exponential search. For resolution, the selection of resolving clauses is exponential. All three use heuristics to contain the exponential processes, however, all are inherently NP, and complete, algorithms.

Set-insertion is a new type of complexity tool, one which exchanges completeness for polynomial efficiency. Some other nice features of set-insertion include:

- It has simple steps which do not rely on search or creativity (in contrast to natural deduction).

- It is very efficient, never generating new forms which increase the search space (in contrast to resolution). All steps reduce a problem through deletion of irrelevant structure.

- It uses virtual forms (i.e., queries) rather than assertions. A positive answer returns a reduced form directly.

- It has powerful pattern-matching and variable counting heuristics which can improve its polynomial performance greatly.

- It returns a minimal form (like algebraic approaches) rather than a Boolean failure.

Boolean Minimization

Here is an example of using insertion for the purpose of Boolean minimization. The key idea which enhances the efficiency of minimization is to avoid both of the tangled transforms Pivot and Distribution. Consider minimizing the following logic expression:

```
(A or B) and ((not (A or C)) or (not (D or (not B))))

((A B) ((A C)(D (B))))                                  transcribe
```

A computationally expensive approach using Distribution:

```
((A B) ((      A C) (      D (B))))
(      (((A B) A C) ((A B) D (B))))                     dis-in
(      (((  B) A C) ((A B) D (B))))                     per A
(      (((  B) A C) (      D (B))))                     abs (A B)
((B)   ((      A C) (      D    )))                     dis-out

       ((B) ((D) (A C)))                                rewrite

B and ((not D) or (not (A C)))                          transcribe
```

A computationally inexpensive approach using both virtual insertion and *virtual distribution*:

```
((A B)     ((        A C) (D (B))))
((A B)     ((^(A B)^ A C) (D (B))))                    ins (A B)
((A B)     ((^(  B)^ A C) (D (B))))                    per A
((A B) (B) ((        A C) (D    )))                    dis-out
(      (B) ((        A C) (D    )))                    abs

       ((B) ((D) (A C)))                               rewrite
```

The distribute-out step is innovative, since it combines a virtual and an actual form of (B) to achieve reduction. So long as Pervasion is used instead of inward Distribution, the potential complexity of using Distribution can be avoided. Its application direction is no longer a guess, it is always outward.

All BL transformations have a virtual counterpart. Here's an algebraic proof of shallow Consensus using virtual absorption.

```
(A B) (C (B)) (A C) = (A B) (C (B))       SHALLOW CONSENSUS

(A B) (C (B)) (^(C (B))^ A C)
(A B) (C (B)) (^(C (B))^ A C)                          ins
(A B) (C (B)) (^(  (B))^ A C)                          per C
(A B) (C (B)) (^    B  ^ A C)                          inv
(A B) (C (B))                                          abs
```

The final shallow Absorption step is

```
(A B) (A ^B^ C) => (A B)
```

Computational Complexity and Virtual Insertion

We begin now to look at known NP tautologies. These problems are not susceptible to the basic set-insertion technique. Characterizing the parens shape of these problems is useful for identifying potential candidates for NP complex tautologies via their symmetries.

For convenience in the following sections, we will call single variables *atoms*, and single variables within a boundary, such as (a), *natoms*. The combination of atoms and natoms is called *literals*. Parens forms containing only literals are called *clauses*.

From complexity theory, we know that any collection of two-literal clauses is tractable and that tautologies which require a three-literal clause are intractable (this is called the 3SAT Problem). From BL, we know that for

virtual insertion, natoms, being more deeply nested, are the most likely location of reduction action.

Intractable Example I: The Pigeon-hole Problem

The *pigeon-hole problem* is cited in the theorem-proving literature as being one that is intractable for tautology checkers. Its formulation reads: "There are N pigeons (first index) and N-1 holes (second index). Not every pigeon can find a separate hole."

The 3-2 pigeon-hole problem consists solely of clauses with two literals, it is thus tractable. The 4-3 pigeon-hole problem, however, is clearly intractable. We first examine the 3-2 pigeon-hole, and then move on to the simplest intractable problem of this type, the 4-3 pigeon-hole.

3-2 Pigeon-hole

```
(and (or 11 12) (or 21 22) (or 31 32)          [each hole has one pigeon]
     (or (not 11)(not 21))                     [each pigeon has one hole]
     (or (not 11)(not 31))
     (or (not 12)(not 22))
     (or (not 12)(not 32))
     (or (not 21)(not 31))
     (or (not 22)(not 32)))
```

As stated, the assertion is false, the expression is unsatisfiable. Transcribing:

```
( (11 12) (21 22) (31 32) ((11)(21)) ((11)(31)) ((12)(22)) ((12)(32)) ((21)(31)) ((22)(32)) )
```

In BL, we can freely remove the outer parens by reversing the value of the form, and we do so, converting an unsatisfiable expression into a tautology. The form below should reduce to a mark, ( ), representing *True*.

```
(11 12) (21 22) (31 32) ((11)(21)) ((11)(31)) ((12)(22)) ((12)(32)) ((21)(31)) ((22)(32))
```

This shape is known as clausal form, each parens has only literals within it, but no deeper nestings. Since there are nine clauses at the top-level of the transcription, it seems like a good idea to use a heuristic to limit the number of insertions, but this is exactly what the pigeon-hole problem thwarts. We must set-insert the entire external context of each clause, i.e. all other clauses, into that clause *for every clause*:

```
( 11  12  ^      (21 22)(31 32)((11)(21))((11)(31))((12)(22))((12)(32))((21)(31))((22)(32))^)
( 21  22  ^(11 12)      (31 32)((11)(21))((11)(31))((12)(22))((12)(32))((21)(31))((22)(32))^)
( 31  32  ^(11 12)(21 22)      ((11)(21))((11)(31))((12)(22))((12)(32))((21)(31))((22)(32))^)
((11)(21) ^(11 12)(21 22)(31 32)         ((11)(31))((12)(22))((12)(32))((21)(31))((22)(32))^)
((11)(31) ^(11 12)(21 22)(31 32)((11)(21))         ((12)(22))((12)(32))((21)(31))((22)(32))^)
((12)(22) ^(11 12)(21 22)(31 32)((11)(21))((11)(31))         ((12)(32))((21)(31))((22)(32))^)
((12)(32) ^(11 12)(21 22)(31 32)((11)(21))((11)(31))((12)(22))         ((21)(31))((22)(32))^)
((21)(31) ^(11 12)(21 22)(31 32)((11)(21))((11)(31))((12)(22))((12)(32))         ((22)(32))^)
((22)(32) ^(11 12)(21 22)(31 32)((11)(21))((11)(31))((12)(22))((12)(32))((21)(31))         ^)
```

We can call upon symmetry to reduce the number of subforms which must be examined to the number of different parens shapes, here two (this is solely a convenience and does not implicate the complexity result):

```
( 11  12  ^      (21 22)(31 32)((11)(21))((11)(31))((12)(22))((12)(32))((21)(31))((22)(32))^)
((11)(21) ^(11 12)(21 22)(31 32)         ((11)(31))((12)(22))((12)(32))((21)(31))((22)(32))^)
```

We follow each clause plus insertion separately. Set-insertion into the two-atom clauses reduces via Pervasion followed by Occlusion, but then the reduction halts.

```
( 11  12  ^      (21 22)(31 32)((11)(21))((11)(31))((12)(22))((12)(32))((21)(31))((22)(32))^)
( 11  12  ^      (21 22)(31 32)((  )(21))((  )(31))((  )(22))((  )(32))((21)(31))((22)(32))^)
( 11  12  ^      (21 22)(31 32)                                        ((21)(31))((22)(32))^)
```

Set-insertion into the two-natom clauses reduces to mark via Pervasion followed by Involution:

```
((11)(21) ^(11 12)(21 22)(31 32)         ((11)(31))((12)(22))((12)(32))((21)(31))((22)(32))^)
((11)(21) ^(11 12)(21 22)(31 32)         (    (31))((12)(22))((12)(32))(    (31))((22)(32))^)
((11)(21) ^(11 12)(21 22)(31 32)              31  ((12)(22))((12)(32))      31  ((22)(32))^)
((11)(21) ^(11 12)(21 22)(   32)              31  ((12)(22))((12)(32))          ((22)(32))^)
((11)(21) ^(11 12)(21 22)(   32)              31  ((12)(22))((12)    )         ((22)    )^)
((11)(21) ^(11 12)(21 22)(   32)              31  ((12)(22))  12                 22      ^)
((11)(21) ^(11   )(21   )(   32)              31  ((12)(22))  12                 22      ^)
(         ^(11   )(21   )(   32)              31  ((12)(22))  12                 22      ^)
(                                                                                  )
```

Note the delicate untangling through successive choreographed exposures of atoms and natoms. The step which deletes the actual (11)(21) natoms calls upon Pervasion in the same space, earlier identified as the rule of Replication. Once the actual natoms are gone, the remaining fragments are deleted as virtual, leaving a mark. This mark then dominates all eight other clauses to complete the proof.

Without a symmetry identification algorithm, nothing less than set-insertion into *every* clause will assure reduction of this type of problem. We now consider an actual intractable problem, the 4-3 pigeon-hole.

4-3 Pigeon-hole

```
(and (or 11 12 13) (or 21 22 23)                          [each hole has one pigeon]
     (or 31 32 33) (or 41 42 43))
     (or (not 11)(not 21)) (or (not 11)(not 31))          [each pigeon has one hole]
     (or (not 11)(not 41)) (or (not 21)(not 31))
     (or (not 21)(not 41)) (or (not 31)(not 41))
     (or (not 12)(not 22)) (or (not 12)(not 32))
     (or (not 12)(not 42)) (or (not 22)(not 32))
     (or (not 22)(not 42)) (or (not 32)(not 42))
     (or (not 13)(not 23)) (or (not 13)(not 33))
     (or (not 13)(not 43)) (or (not 23)(not 33))
     (or (not 23)(not 43)) (or (not 33)(not 43)))
```

Transcribing and dropping the outer parens to make the clausal form *True*:

```
(11 12 13) (21 22 23) (31 32 33) (41 42 43)
((11)(21)) ((11)(31)) ((11)(41)) ((12)(22)) ((12)(32)) ((12)(42))
((13)(23)) ((13)(33)) ((13)(43)) ((21)(31)) ((21)(41)) ((22)(32))
((22)(42)) ((23)(33)) ((23)(43)) ((31)(41)) ((32)(42)) ((33)(43))
```

The slight change of having three-atom clauses makes the difference between a tractable and an intractable tautology. Twenty-two clauses are needed to express the 4-3 pigeon-hole problem. For convenience we call again upon symmetry to examine set-insertion for only two.

The first insertion into a three-atom clause proceeds as in the 3-2 pigeon-hole problem, and halts prior to complete reduction:

```
(11 12 13 ^(21 22 23)(31 32 33)(41 42 43)
   ((11)(21))((11)(31))((11)(41))((12)(22))((12)(32))((12)(42))((13)(23))((13)(33))((13)(43))
   ((21)(31))((21)(41))((22)(32))((22)(42))((23)(33))((23)(43))((31)(41))((32)(42))((33)(43))^)

(11 12 13 ^(21 22 23)(31 32 33)(41 42 43)
   ((  )(21))((  )(31))((  )(41))((  )(22))((  )(32))((  )(42))((  )(23))((  )(33))((  )(43))
   ((21)(31))((21)(41))((22)(32))((22)(42))((23)(33))((23)(43))((31)(41))((32)(42))((33)(43))^)

(11 12 13 ^(21 22 23)(31 32 33)(41 42 43)
   ((21)(31))((21)(41))((22)(32))((22)(42))((23)(33))((23)(43))((31)(41))((32)(42))((33)(43))^)
```

The failure to reduce should not be surprising, since the three atoms which share space with the insertion do not permit the two-natom insertion clauses

to reduce to a mark. We proceed with set-insertion into the other clausal shape, those with two natoms:

```
           ((11)(31))((11)(41))((12)(22))((12)(32))((12)(42))((13)(23))((13)(33))((13)(43))
  ((21)(31))((21)(41))((22)(32))((22)(42))((23)(33))((23)(43))((31)(41))((32)(42))((33)(43))^)

((11)(21) ^(11 12 13)(21 22 23)(31 32 33)(41 42 43)
            (    (31))(    (41))((12)(22))((12)(32))((12)(42))((13)(23))((13)(33))((13)(43))
  (    (31))(    (41))((22)(32))((22)(42))((23)(33))((23)(43))((31)(41))((32)(42))((33)(43))^)

((11)(21) ^(11 12 13)(21 22 23)(31 32 33)(41 42 43)
                  31        41  ((12)(22))((12)(32))((12)(42))((13)(23))((13)(33))((13)(43))
        31        41  ((22)(32))((22)(42))((23)(33))((23)(43))((31)(41))((32)(42))((33)(43))^)

((11)(21) ^(11 12 13)(21 22 23)(   32 33)(   42 43)
                  31        41  ((12)(22))((12)(32))((12)(42))((13)(23))((13)(33))((13)(43))
                      ((22)(32))((22)(42))((23)(33))((23)(43))((   )(  ))((32)(42))((33)(43))^)

((11)(21) ^(11 12 13)(21 22 23)(   32 33)(   42 43)
                  31        41  ((12)(22))((12)(32))((12)(42))((13)(23))((13)(33))((13)(43))
                      ((22)(32))((22)(42))((23)(33))((23)(43))          ((32)(42))((33)(43))^)
```

Unfortunately, removing new atoms does not create new natoms, but rather new two-atom subforms. Here is the crux of why three-atom clauses make an intractable tautology. We might try to insert into the next deeper level. That insertion deletes only one atom since 11 and 21 occur only once in the insertion. Unfortunately, both occur in separate three-atom clauses, again halting the reduction.

A Case Analysis

We can verify that the above subform, with insertions, is indeed a tautology by taking an exponential step, to consider two cases, one in which, say, 32 is <void> and one in which 32 is a mark. The variable 32 is a good choice because it is inside one of the newly formed two-atom clauses. The case analysis first deletes 32 and reduces, then marks 32 and reduces. Both forms reduce to mark, i.e. they are both *True*. Thus the subform is *True*, and by Occlusion, the entire expression is *True*.

Substitution of <void>

Void-substitution for (i.e., deleting) variable 32 leads to reduction success:

```
((11)(21) ^(11 12 13)(21 22 23)(      33)(   42 43)
                  31        41  ((12)(22))((12)(  ))((12)(42))((13)(23))((13)(33))((13)(43))
                      ((22)(  ))((22)(42))((23)(33))((23)(43))          ((  )(42))((33)(43))^)

((11)(21) ^(11 12 13)(21 22 23)(      33)(   42 43)
                  31        41  ((12)(22))          ((12)(42))((13)(23))((13)    )((13)(43))
                                ((22)(42))((23)    )((23)(43))                    (    (43))^)

((11)(21) ^(11 12 13)(21 22 23)(      33)(   42 43)
                  31        41  ((12)(22))          ((12)(42))((13)(23))  13      ((13)(43))
                                ((22)(42))  23      ((23)(43))                          43  ^)

((11)(21) ^(11 12   )(21 22   )(      33)(   42   )
                  31        41  ((12)(22))          ((12)(42))((  )(  ))  13      ((  )(  ))
                                ((22)(42))  23      ((  )(  ))                          43  ^)
```

Rewriting the current reduction, and continuing:

```
      ((11)(21) ^(11 12)(21 22)(33)(42) 13 23 31 41 43 ((12)(22))((12)(42))((22)(42))^)
      ((11)(21) ^(11 12)(21 22)(33)(42) 13 23 31 41 43 ((12)(22))((12)    )((22)    )^)
      ((11)(21) ^(11 12)(21 22)(33)(42) 13 23 31 41 43 ((12)(22))  12        22      ^)
      ((11)(21) ^(11   )(21   )(33)(42) 13 23 31 41 43 ((  )(  ))  12        22      ^)
      (         ^(11   )(21   )(33)(42) 13 23 31 41 43             12        22      ^)
      (                                                                               )
```

Now we return to the case in which 32 is a mark, in the process expending exponential effort.

Substitution of mark

The substitution of mark for 32 also succeeds:

```
((11)(21) ^(11 12 13)(21 22 23)(   () 33)(   42 43)
                  31        41  ((12)(22))((12)(()))((12)(42))((13)(23))((13)(33))((13)(43))
                      ((22)(()))((22)(42))((23)(33))((23)(43))          ((())(42))((33)(43))^)

((11)(21) ^(11 12 13)(21 22 23)         (   42 43)
                  31        41  ((12)(22))((12)    )((12)(42))((13)(23))((13)(33))((13)(43))
                      ((22)    )((22)(42))((23)(33))((23)(43))          (    (42))((33)(43))^)

((11)(21) ^(11 12 13)(21 22 23)         (   42 43)
                  31        41  ((12)(22))  12      ((12)(42))((13)(23))((13)(33))((13)(43))
                        22      ((22)(42))((23)(33))((23)(43))               42  ((33)(43))^)

((11)(21) ^(11    13)(21    23)         (      43)
                  31        41  ((  )(  ))  12      ((  )(  ))((13)(23))((13)(33))((13)    )
                        22      ((  )(  ))((23)(33))((23)    )               42  ((33)    )^)
```

Rewriting and continuing:

```
((11)(21) ^(11 13)(21 23)(43) 12 13 22 23 31 33 41 42 ((13)(23))((13)(33))((23)(33))^)
((11)(21) ^(11   )(21   )(43) 12 13 22 23 31 33 41 42 ((  )(  ))((  )(  ))((  )(  ))^)
(         ^(11   )(21   )(43) 12 13 22 23 31 33 41 42                               ^)
(                                                                                    )
```

Thus we have succeeded not only in reducing this problem, but also in identifying a problem which may require exponential effort to reduce. We are close to the boundary between tractable and intractable tautologies.

Intractability

In order to extend set-insertion so that it will reduce this problem, we need to find a clever way to reduce the insertion subproblem above, prior to the exponential split:

```
((11)(21) ^31 41 (11 12 13)(21 22 23)(32 33)(42 43)((12)(22))((12)(32))((12)(42))((13)(23))
          ((13)(33))((13)(43))((22)(32))((22)(42))((23)(33))((23)(43))((32)(42))((33)(43))^)
```

Reducing this subproblem to <void>, that is, reducing the insertion to a mark, would not succeed in demonstrating a tautology. By symmetry, all two-natom clauses in the original problem would then erase, leaving only three-atom clauses. A problem stated in clausal form which has only atoms, or only natoms, is not a tautology. This general observation rests upon a simple principle of Pervasion: to reduce a clause to a mark, at least one variable must occur at both odd and even depths. Above, we know that the insertion itself must reduce to a subform which contains either (11) or (21). However, we are now no longer in the domain of identifying a tautology, rather this is a problem in Boolean minimization, the "intractable intractable" problem.

In summary: Three-atom forms will halt reduction via Pervasion, while clever approaches will be stymied by Boolean minimization rather than by tautology detection.

To provide an appreciation about how exponential problems get out of hand, consider the effect of increasing the number of pigeons. Going from 3 to 4 pigeons made tractable two-atom clauses into intractable three-atom clauses. In going to 5 pigeons, the same clauses will contain four atoms. In the prior reduction of the three-atom clauses, one variable needed to be removed via case analysis, which doubled the work. Four-atom clauses will reduce to three-atom clauses when one variable is removed. In this case, the doubled effort does not provide a solution, instead it leaves another problem (going from three-atom clauses to two-atom clauses) which we already know doubles the effort yet again. Thus every new pigeon beyond three will double the effort needed to reduce the pigeon-hole tautology.

However, the central point of this entire exploration is that, using BL, the type of problem that requires exponential effort must indeed be very tangled, and in a quite symmetrical way, with appropriate occurrences of appropriate variables in just the right places and just the right depth of nesting. Simply by counting variable occurrences after Pervasion reduction fails, we might know the chances that we have a tangled tautology.

We will return to this problem after identifying how to extend insertion to intractable problems.

Intractable Example II: Three Coloring a Tetrahedron

Symmetries in the representation of the problem of three coloring a tetrahedron provide nice properties in that there are very few natoms. This problem is known to be NP. Its formulation is: "Assign three colors to each of the six edges of a tetrahedron so that no vertex has two edges of the same color."

The coloring problem is encoded by stating the constraints on colors {B,G,R} and on the vertices {1,2,3,4} of a tetrahedron. Clauses that reference the same color, (B1 B2) for example, state that each edge ends in the same color; more literally the clauses state that any edge is one color. Clauses which reference the same vertex, (B1 G1) for example, state that edges terminating at a specific vertex must be of a different color. Here is the parens encoded problem:

```
(B1 B2)(B1 B3)(B1 B4)(B2 B3)(B2 B4)(B3 B4)    [edges are the same color]
(G1 G2)(G1 G3)(G1 G4)(G2 G3)(G2 G4)(G3 G4)
(R1 R2)(R1 R3)(R1 R4)(R2 R3)(R2 R4)(R3 R4)
(B1 G1)(B1 R1)(G1 R1)((B1)(G1)(R1))           [each vertex has a color exactly once]
(B2 G2)(B2 R2)(G2 R2)((B2)(G2)(R2))
(B3 G3)(B3 R3)(G3 R3)((B3)(G3)(R3))
(B4 G4)(B4 R4)(G4 R4)((B4)(G4)(R4))
```

For clausal forms such as the one above, we use the heuristic that reduction terminates with insertion into clauses with natoms, as occurred in the pigeon-hole problem. The absence of natoms in the predominance of the clauses in this problem keeps them from interacting and thus keeps reduction well constrained. What kind of tangle is holding up this form? We know that the eventual collapse will be caused by one of the clauses becoming a mark, ( ).

Partial Reduction

Simple set-insertion yields the following reduction: clauses referring to the same vertex vanish. For example, consider the single clause (B1 G1) when all other clauses are inserted into it:

```
(B1 G1 ^(B1 B2)(B1 B3)(B1 B4)(B2 B3)(B2 B4)(B3 B4)
       (G1 G2)(G1 G3)(G1 G4)(G2 G3)(G2 G4)(G3 G4)
       (R1 R2)(R1 R3)(R1 R4)(R2 R3)(R2 R4)(R3 R4)
              (B1 R1)(G1 R1)((B1)(G1)(R1))
       (B2 G2)(B2 R2)(G2 R2)((B2)(G2)(R2))
       (B3 G3)(B3 R3)(G3 R3)((B3)(G3)(R3))
       (B4 G4)(B4 R4)(G4 R4)((B4)(G4)(R4))^)
```

Extracting occurrences of the atoms B1 and G1 generates new natoms, which in turn free up new atoms, etc., a pattern we have seen before.

```
(B1 G1 ^(   B2)(   B3)(   B4)(B2 B3)(B2 B4)(B3 B4)
       (   G2)(   G3)(   G4)(G2 G3)(G2 G4)(G3 G4)
       (R1 R2)(R1 R3)(R1 R4)(R2 R3)(R2 R4)(R3 R4)
              (   R1)(   R1)((B1)(G1)(R1))
       (B2 G2)(B2 R2)(G2 R2)((B2)(G2)(R2))
       (B3 G3)(B3 R3)(G3 R3)((B3)(G3)(R3))
       (B4 G4)(B4 R4)(G4 R4)((B4)(G4)(R4))^)

(B1 G1 ^(   B2)(   B3)(   B4)
       (   G2)(   G3)(   G4)
                             (R2 R3)(R2 R4)(R3 R4)
              (   R1)        ((B1)(G1)    )
                             (        (R2))
                             (        (R3))
                             (        (R4))^)

(B1 G1 ^(   B2)(   B3)(   B4)
       (   G2)(   G3)(   G4)
                             (     )(     )(     )
              (   R1)        ((B1)(G1)    )
                                      R2
                                      R3
                                      R4  ^)
(B1 G1  ^(  )^ )
```

Thus, Occlusion causes the entire clause to vanish. All clauses with a structure symmetrical to (B1 G1) are void-equivalent, however this does not help in reducing the problem to a mark.

More Set Insertions

Now we consider the effect of set insertion on clauses referencing the same color, such as (B1 B2). Although 12 two-atom clauses have been shown above to be void-equivalent, they are still essential during set-insertion, since they provide contextual information for the reduction.

```
(B1 B2 ^      (B1 B3)(B1 B4)(B2 B3)(B2 B4)(B3 B4)
       (G1 G2)(G1 G3)(G1 G4)(G2 G3)(G2 G4)(G3 G4)
       (R1 R2)(R1 R3)(R1 R4)(R2 R3)(R2 R4)(R3 R4)
       (B1 G1)(B1 R1)(G1 R1)((B1)(G1)(R1))
       (B2 G2)(B2 R2)(G2 R2)((B2)(G2)(R2))
       (B3 G3)(B3 R3)(G3 R3)((B3)(G3)(R3))
       (B4 G4)(B4 R4)(G4 R4)((B4)(G4)(R4))^)
```

We rewrite to conserve display space. The two atoms create new natoms, which in turn pervade the three-natom clauses to create the reduction:

```
(B1 B2 ^(   B3)(   B4)(   B3)(   B4)(B3 B4)(G1 G2)(G1 G3)(G1 G4)(G2 G3)(G2 G4)(G3 G4)(R1 R2)(R1 R3)
        (R1 R4)(R2 R3)(R2 R4)(R3 R4)(   G1)(   R1)(G1 R1)(   G2)(   R2)(G2 R2)(B3 G3)(B3 R3)(G3 R3)
        (B4 G4)(B4 R4)(G4 R4)((B1)(G1)(R1))((B2)(G2)(R2))((B3)(G3)(R3))((B4)(G4)(R4))^)

(B1 B2 ^(B3)(B4)(G1)(R1)(G2)(R2) (G3 G4)(R3 R4)(G3 R3)(G4 R4)((B1))((B2))((G3)(R3))((G4)(R4))^)
(B1 B2 ^(B3)(B4)(G1)(R1)(G2)(R2) (G3 G4)(R3 R4)(G3 R3)(G4 R4)  B1    B2  ((G3)(R3))((G4)(R4))^)
(      ^(B3)(B4)(G1)(R1)(G2)(R2) (G3 G4)(R3 R4)(G3 R3)(G4 R4)  B1    B2  ((G3)(R3))((G4)(R4))^)
(                                                                                            )
```

Thus, by symmetry, all two-atom color clauses are void-equivalent. Note that the variables B1 and B2 were not removed from their natom positions inside the three-natom clauses. Removing them would have obscured the potential reduction.

Demonstrating that even more clauses are void-equivalent does not result in reduction to a mark. We might expect reduction when clauses are set-inserted into those other clauses containing natoms. We explore only one, ((B1)(G1)(R1)), inserting directly into the natom (R1):

```
((B1)(G1)(R1 ^(B1 B2)(B1 B3)(B1 B4)(B2 B3)(B2 B4)(B3 B4)(G1 G2)(G1 G3)(G1 G4)(G2 G3)(G2 G4)(G3 G4)
              (R1 R2)(R1 R3)(R1 R4)(R2 R3)(R2 R4)(R3 R4)(B1 G1)(B1 R1)(G1 R1)(B2 G2)(B2 R2)(G2 R2)
              ((B2)(G2)(R2))(B3 G3)(B3 R3)(G3 R3)((B3)(G3)(R3))(B4 G4)(B4 R4)(G4 R4)((B4)(G4)(R4))^))

((B1)(G1)(R1 ^(B1 B2)(B1 B3)(B1 B4)(B2 B3)(B2 B4)(B3 B4)(G1 G2)(G1 G3)(G1 G4)(G2 G3)(G2 G4)(G3 G4)
              (   R2)(   R3)(   R4)(R2 R3)(R2 R4)(R3 R4)(B1 G1)(B1   )(G1   )(B2 G2)(B2 R2)(G2 R2)
              ((B2)(G2)(R2))(B3 G3)(B3 R3)(G3 R3)((B3)(G3)(R3))(B4 G4)(B4 R4)(G4 R4)((B4)(G4)(R4))^))

((B1)(G1)(R1 ^                    (B2 B3)(B2 B4)(B3 B4)                    (G2 G3)(G2 G4)(G3 G4)
              (   R2)(   R3)(   R4)                          (B1   )(G1   )(B2 G2)
              ((B2)(G2)    )(B3 G3)             ((B3)(G3)    )(B4 G4)             ((B4)(G4)    )^))
```

This form does not reduce. It appears that virtual insertion cannot reduce this problem.

Success in Failure

By inserting into the three-natom clauses, the insertion reduced to one which makes no reference to vertex 1. After pervading (B1)(G1), the insertion directly above is:

```
((B1)(G1)(R1 ^(B2 B3)(B2 B4)(B3 B4)(G2 G3)(G2 G4)(G3 G4)(B2 G2)(B3 G3)(B4 G4)
             (R2)(R3)(R4)((B2)(G2))((B3)(G3))((B4)(G4))^ ))
```

Should the insertion in the above subproblem reduce to a mark, it would succeed in reducing the (R1) natom to <void>. That is, we might ask: Is the above *insertion* a tautology? We can recursively enter the same virtual insertion algorithm to obtain an answer.

```
             (B2 B3)(B2 B4)(B3 B4)(G2 G3)(G2 G4)(G3 G4)(B2 G2)(B3 G3)(B4 G4)
             (R2)(R3)(R4)((B2)(G2))((B3)(G3))((B4)(G4))
```

It is easy to see that the natoms (R2)(R3)(R4) will not participate in a reduction, they can be omitted without loss:

```
(B2 B3)(B2 B4)(B3 B4)(G2 G3)(G2 G4)(G3 G4)(B2 G2)(B3 G3)(B4 G4)((B2)(G2))((B3)(G3))((B4)(G4))
```

We now apply set-insertion to the reduced insertion itself, inserting into one of the clauses with atoms:

```
(B2 B3 ^(B2 B4)(B3 B4)(G2 G3)(G2 G4)(G3 G4)(B2 G2)(B3 G3)(B4 G4)((B2)(G2))((B3)(G3))((B4)(G4))^)
(B2 B3 ^(   B4)(   B4)(G2 G3)(G2 G4)(G3 G4)(   G2)(   G3)(B4 G4)((B2)(G2))((B3)(G3))((B4)(G4))^)
(B2 B3 ^(   B4)                           (   G2)(   G3)       ((B2)    )((B3)    )(    (G4))^)
(B2 B3 ^(   B4)                           (   G2)(   G3)         B2        B3           G4  ^)
(      ^(   B4)                           (   G2)(   G3)         B2        B3           G4  ^)
(                                                                                           )
```

The void-equivalence of the two-atom clauses is consistent with what was found before. Next we explore the second insertion type, into natom clauses:

```
((B2)(G2 ^(B2 B3)(B2 B4)(B3 B4)(G2 G3)(G2 G4)(G3 G4)(B2 G2)(B3 G3)(B4 G4)((B3)(G3))((B4)(G4))^))
((B2)(G2 ^(B2 B3)(B2 B4)(B3 B4)(   G3)(   G4)(G3 G4)(B2   )(B3 G3)(B4 G4)((B3)(G3))((B4)(G4))^))
((B2)(G2 ^             (B3 B4)(   G3)(   G4)       (B2   )             ((B3)    )((B4)    )^))
((B2)(G2 ^             (B3 B4)(   G3)(   G4)       (B2   )               B3        B4      ^))
((B2)(G2 ^             (     )(   G3)(   G4)       (B2   )               B3        B4      ^))
((B2)                                                                                       )
  B2
```

This succeeds in removing G2. Substituting B2 for its original clause ((B2)(G2)) results in the following subproblem:

```
(B2 B3)(B2 B4)(B3 B4)(G2 G3)(G2 G4)(G3 G4)(B2 G2)(B3 G3)(B4 G4)  B2      ((B3)(G3))((B4)(G4))
(   B3)(   B4)(B3 B4)(G2 G3)(G2 G4)(G3 G4)(   G2)(B3 G3)(B4 G4)  B2      ((B3)(G3))((B4)(G4))
(   B3)(   B4)                     (G3 G4)(   G2)                B2      (    (G3))(    (G4))
(   B3)(   B4)                     (G3 G4)(   G2)                B2           G3        G4
(   B3)(   B4)                     (     )(   G2)                B2           G3        G4
                                   (     )
```

The clausal reduction propagates to reduce the entire insertion to mark. We have demonstrated that the following form is a tautology:

```
            (B2 B3)(B2 B4)(B3 B4)(G2 G3)(G2 G4)(G3 G4)(B2 G2)(B3 G3)(B4 G4)
            (R2)(R3)(R4)((B2)(G2))((B3)(G3))((B4)(G4))
```

Returning from the recursion, we can substitute this result into the context it came from:

```
((B1)(G1)(R1 ^(B2 B3)(B2 B4)(B3 B4)(G2 G3)(G2 G4)(G3 G4)(B2 G2)(B3 G3)(B4 G4)
            (R2)(R3)(R4)((B2)(G2))((B3)(G3))((B4)(G4))^ ))

      ((B1)(G1)(R1 ^( )^))
      ((B1)(G1)         )
```

Calling upon symmetry, we can also eliminate (B1) and (G1). Only (B1) is demonstrated:

```
((G1)(R1)(B1 ^(B1 B2)(B1 B3)(B1 B4)(B2 B3)(B2 B4)(B3 B4)(G1 G2)(G1 G3)(G1 G4)(G2 G3)(G2 G4)(G3 G4)
            (R1 R2)(R1 R3)(R1 R4)(R2 R3)(R2 R4)(R3 R4)(B1 G1)(B1 R1)(G1 R1)(B2 G2)(B2 R2)(G2 R2)
            ((B2)(G2)(R2))(B3 G3)(B3 R3)(G3 R3)((B3)(G3)(R3))(B4 G4)(B4 R4)(G4 R4)((B4)(G4)(R4))^))

((G1)(R1)(B1 ^(   B2)(   B3)(   B4)(B2 B3)(B2 B4)(B3 B4)(G1 G2)(G1 G3)(G1 G4)(G2 G3)(G2 G4)(G3 G4)
            (R1 R2)(R1 R3)(R1 R4)(R2 R3)(R2 R4)(R3 R4)(   G1)(   R1)(G1 R1)(B2 G2)(B2 R2)(G2 R2)
            ((B2)(G2)(R2))(B3 G3)(B3 R3)(G3 R3)((B3)(G3)(R3))(B4 G4)(B4 R4)(G4 R4)((B4)(G4)(R4))^))

((G1)(R1)(B1 ^(   B2)(   B3)(   B4)                                        (G2 G3)(G2 G4)(G3 G4)
                                  (R2 R3)(R2 R4)(R3 R4)(   G1)(   R1)                  (G2 R2)
            (    (G2)(R2))             (G3 R3)(    (G3)(R3))             (G4 R4)(    (G4)(R4))^))
```

We now take another recursive step, determining whether or not the new insertion itself is a tautology. Rewriting the insertion while deleting the natoms which cannot be involved in further reductions, yields:

```
(G2 G3)(G2 G4)(G3 G4)(R2 R3)(R2 R4)(R3 R4)(G2 R2)(G3 R3)(G4 R4)((G2)(R2))((G3)(R3))((G4)(R4))
```

Now inserting into a two-natom clause of this subproblem:

```
((G2)(R2 ^(G2 G3)(G2 G4)(G3 G4)(R2 R3)(R2 R4)(R3 R4)(G2 R2)(G3 R3)(G4 R4)((G3)(R3))((G4)(R4))^))
((G2)(R2 ^(G2 G3)(G2 G4)(G3 G4)(   R3)(   R4)(R3 R4)(G2   )(G3 R3)(G4 R4)((G3)(R3))((G4)(R4))^))
((G2)(R2 ^             (G3 G4)(   R3)(   R4)                          ((G3)    )((G4)    )^))
((G2)(R2 ^             (G3 G4)(   R3)(   R4)                            G3        G4      ^))
((G2)(R2 ^             (     )(   R3)(   R4)                            G3        G4      ^))
((G2)                                                                                       )
  G2
```

Substituting the result, and further reducing:

```
(G2 G3)(G2 G4)(G3 G4)(R2 R3)(R2 R4)(R3 R4)(G2 R2)(G3 R3)(G4 R4)  G2      ((G3)(R3))((G4)(R4))
(   G3)(   G4)(G3 G4)(R2 R3)(R2 R4)(R3 R4)(   R2)(G3 R3)(G4 R4)  G2      ((G3)(R3))((G4)(R4))
(   G3)(   G4)                     (R3 R4)(   R2)                G2      (    (R3))(    (R4))
(   G3)(   G4)                     (R3 R4)(   R2)                G2           R3        R4
(   G3)(   G4)                     (     )(   R2)                G2           R3        R4
                                   (     )
```

And finally, returning from the recursion with the current result:

```
      ((G1)(R1)(B1 ^( )^))
      ((G1)(R1)         )
```

This demonstrates that all the natoms in one of the three-natom clauses in the original problem are void-equivalent, leaving:

```
            (B1 B2)(B1 B3)(B1 B4)(B2 B3)(B2 B4)(B3 B4)
            (G1 G2)(G1 G3)(G1 G4)(G2 G3)(G2 G4)(G3 G4)
            (R1 R2)(R1 R3)(R1 R4)(R2 R3)(R2 R4)(R3 R4)
            (B1 G1)(B1 R1)(G1 R1)(           )
            (B2 G2)(B2 R2)(G2 R2)((B2)(G2)(R2))
            (B3 G3)(B3 R3)(G3 R3)((B3)(G3)(R3))
            (B4 G4)(B4 R4)(G4 R4)((B4)(G4)(R4))
```

The mark dominates all other clauses, and the proof is complete. Note that the fact that all two-atom clauses are void-equivalent is irrelevant to the result.

Is this proof tractable? Certainly not. By taking the recursive step, we have begun a search process. Basically we do not know which of the many possible insertion fragments, generated by insertions into different clauses, will succeed and which will fail. Thus, each insertion may spawn several options to be explored, and the successful one is not known in advance. Each option , as well, may spawn yet other suboptions. This circumstance is

significantly different than that of the original set-insertions which are bounded by the number of nested parens, and do not themselves generate the need for more reduction effort. As the number of clauses and variables increases, the recursive insertion technique enters into more and more recursions, each branching exponentially.

Now we can say exactly where the boundary between P and NP effort is:

If set-insertion must be applied recursively,
a tautology takes exponential effort to reduce.

This operational definition provides a new complexity analysis tool, one that can distinguish between problems which are intractable and those which are not.

The 4-3 Pigeon-hole Revisited

For completion, we now apply the recursive set-insertion technique to the 4-3 pigeon-hole problem which previously required case analysis to reduce. Although this example is of exponential complexity, it serves to illustrate the powerful heuristic filters available for NP problems expressed in parens form.

Recurring on an Insertion

We left the 4-3 pigeon-hole while working on the following fragment. At the time, we also left polynomial simplicity by applying case analysis to this fragment.

```
((11)(21) ^31 41 (11 12 13)(21 22 23)(32 33)(42 43)((12)(22))((12)(32))((12)(42))((13)(23))
          ((13)(33))((13)(43))((22)(32))((22)(42))((23)(33))((23)(43))((32)(42))((33)(43))^)
```

We argued that this subform could not erase, since it is the source of natoms in the problem statement; we must be able to reduce it to a mark or to a single variable. Therefore the insertion must contain either (11) or (12), preferably both. Taking the insertion as a new problem, we proceed with the recursive reduction:

```
31 41 (11 12 13)(21 22 23)(32 33)(42 43)((12)(22))((12)(32))((12)(42))((13)(23))
((13)(33))((13)(43))((22)(32))((22)(42))((23)(33))((23)(43))((32)(42))((33)(43))
```

The first observation is that the single atoms cannot effect the result. We also observe that the only route to success is to reduce one of the three-atom clauses, since that is the only way to end up with the appropriate *natom*, here either (11) or (21). Note that there is no expectation of being able to avoid reducing the three-atom clauses, since we know that the problem will not yield to a simple solution. Selecting the first two-natom clause, set-inserting and reducing:

```
((12)(22) ^(11 12 13)(21 22 23)(32 33)(42 43)((12)(32))((12)(42))((13)(23))((13)(33))
          ((13)(43))((22)(32))((22)(42))((23)(33))((23)(43))((32)(42))((33)(43))^ )

((12)(22) ^(11 12 13)(21 22 23)(32 33)(42 43)(    (32))(    (42))((13)(23))((13)(33))
          ((13)(43))(    (32))(    (42))((23)(33))((23)(43))((32)(42))((33)(43))^ )

((12)(22) ^(11 12 13)(21 22 23)(32 33)(42 43)      32        42  ((13)(23))((13)(33))
          ((13)(43))      32        42  ((23)(33))((23)(43))((32)(42))((33)(43))^ )

((12)(22) ^(11 12 13)(21 22 23)(   33)(   43)      32        42  ((13)(23))((13)(33))
          ((13)(43))                   ((23)(33))((23)(43))((  )(  ))((33)(43))^ )

((12)(22) ^(11 12 13)(21 22 23)(   33)(   43)      32        42  ((13)(23))((13)    )
          ((13)    )                   ((23)    )((23)    )          (        )^ )
```

We have succeeded in demonstrating that this two-natom clause is void-equivalent, which does not achieve the goal. We demonstrate the reduction of the next two-natom clause:

```
((12)(32) ^(11 12 13)(21 22 23)(32 33)(42 43)((12)(22))((12)(42))((13)(23))((13)(33))
          ((13)(43))((22)(32))((22)(42))((23)(33))((23)(43))((32)(42))((33)(43))^ )

((12)(32) ^(11 12 13)(21 22 23)(32 33)(42 43)(    (22))(    (42))((13)(23))((13)(33))
          ((13)(43))((22)    )((22)(42))((23)(33))((23)(43))(    (42))((33)(43))^ )

((12)(32) ^(11 12 13)(21 22 23)(32 33)(42 43)      22        42  ((13)(23))((13)(33))
          ((13)(43))  22      ((22)(42))((23)(33))((23)(43))      42  ((33)(43))^ )

((12)(32) ^(11 12 13)(21    23)(32 33)(   43)      22        42  ((13)(23))((13)(33))
          ((13)(43))          ((  )(  ))((23)(33))((23)(43))          ((33)(43))^ )
```

Rewriting and continuing:

```
((12)(32) ^22 42 (11 12 13)(21 23)(32 33)(43)((13)(23))((13)(33))((13))((23)(33))((23))((33))^ )
((12)(32) ^22 42 (11 12 13)(21 23)(32 33)(43)((13)(23))((13)(33))  13  ((23)(33))  23    33  ^ )
((12)(32) ^22 42 (11 12   )(21   )(32   )(43)((  )(  ))((  )(  ))  13  ((  )(  ))  23    33  ^ )
((12)     ^22 42 (11 12   )(21   )(32   )(43)                      13              23    33  ^ )
((12)                                                                                        )
  12
```

This result has reduced a two-natom clause to a single atom. We substitute into the context it came from:

```
          31 41 (11 12 13)(21 22 23)(32 33)(42 43)((12)(22))  12      ((12)(42))((13)(23))
          ((13)(33))((13)(43))((22)(32))((22)(42))((23)(33))((23)(43))((32)(42))((33)(43))
```

This single result is not sufficient to complete the reduction. In a complete set-insertion, each of the twelve two-atom clauses would return a result concurrently. Another processing strategy is to incorporate single

reductions one at a time and reduce sequentially. We illustrate both approaches with the next two-natom clause that results in reduction.

Parallel Reduction

```
((13)(33) ^(11 12 13)(21 22 23)(32 33)(42 43)((12)(22))((12)(32))((12)(42))((13)(23))
                  ((13)(43))((22)(32))((22)(42))((23)(33))((23)(43))((32)(42))((33)(43))^)

((13)(33) ^(11 12 13)(21 22 23)(32 33)(42 43)((12)(22))((12)(32))((12)(42))(    (23))
                  (    (43))((22)(32))((22)(42))((23)    )((23)(43))((32)(42))(    (43))^)

((13)(33) ^(11 12 13)(21 22 23)(32 33)(42 43)((12)(22))((12)(32))((12)(42))      23
                        43  ((22)(32))((22)(42))  23      ((23)(43))((32)(42))      43  ^)

((13)(33) ^(11 12 13)(21 22   )(32 33)(42   )((12)(22))((12)(32))((12)(42))      23
                        43  ((22)(32))((22)(42))          ((  )(  ))((32)(42))         ^)
```

Rewriting and continuing:

```
((13)(33) ^23 43 (11 12 13)(21 22)(32 33)(42)((12)(22))((12)(32))((12))((22)(32))((22))((32))^)
((13)(33) ^23 43 (11 12 13)(21 22)(32 33)(42)((12)(22))((12)(32))  12  ((22)(32))  22    32  ^)
((13)(33) ^23 43 (11    13)(21   )(   33)(42)((  )(  ))((  )(  ))  12  ((  )(  ))  22    32  ^)
((13)     ^23 43 (11    13)(21   )(   33)(42)                      12              22    32  ^)
((13)                                                                                      )
  13
```

Sequential Reduction

```
((13)(33) ^(11 12 13)(21 22 23)(32 33)(42 43)((12)(22))  12      ((12)(42))((13)(23))
                  ((13)(43))((22)(32))((22)(42))((23)(33))((23)(43))((32)(42))((33)(43))^)

((13)(33) ^(11    13)(21 22 23)(32 33)(42 43)((  )(22))  12      ((  )(42))(    (23))
                     (    (43))((22)(32))((22)(42))((23)    )((23)(43))((32)(42))(    (43))^)
```

Rewriting and continuing:

```
((13)(33) ^12 23 43 (11 13)(21 22 23)(32 33)(42 43)((22)(32))((22)(42))((23)(43))((32)(42))^)
((13)(33) ^12 23 43 (11 13)(21 22   )(32 33)(42   )((22)(32))((22)(42))((  )(  ))((32)(42))^)
((13)(33) ^12 23 43 (11 13)(21 22   )(32 33)(42   )((22)(32))((22)    )          ((32)    )^)
((13)(33) ^12 23 43 (11 13)(21 22   )(32 33)(42   )((22)(32))  22                  32      ^)
((13)(33) ^12 23 43 (11 13)(21      )(   33)(42   )((  )(  ))  22                  32      ^)
((13)     ^12 23 43 (11 13)(21      )(   33)(42   )            22                  32      ^)
((13)                                                                                    )
  13
```

We return the parallel result to the current subproblem, having now provided two clauses which have reduced to atoms, and proceed with the reductions that those atoms generate:

```
((11)(21) ^31 41 (11 12 13)(21 22 23)(32 33)(42 43)  12      ((12)(32))((12)(42))((13)(23))
             13      ((13)(43))((22)(32))((22)(42))((23)(33))((23)(43))((32)(42))((33)(43))^)

((11)(21) ^31 41 (11      )(21 22 23)(32 33)(42 43)  12      ((  )(32))((  )(42))((  )(23))
             13      ((  )(43))((22)(32))((22)(42))((23)(33))((23)(43))((32)(42))((33)(43))^)

(    (21) ^31 41 (11      )(21 22 23)(32 33)(42 43)  12
             13               ((22)(32))((22)(42))((23)(33))((23)(43))((32)(42))((33)(43))^)

(    (21)                                                                              )
      21
```

We can now exit the entire recursion, returning to the original 4-3 pigeon-hole problem with a simplification

```
before: (11 12 13) (21 22 23) (31 32 33) (41 42 43)
        ((11)(21)) ((11)(31)) ((11)(41)) ((12)(22)) ((12)(32)) ((12)(42))
        ((13)(23)) ((13)(33)) ((13)(43)) ((21)(31)) ((21)(41)) ((22)(32))
        ((22)(42)) ((23)(33)) ((23)(43)) ((31)(41)) ((32)(42)) ((33)(43))

after:  (11 12 13) (21 22 23) (31 32 33) (41 42 43)
              21   ((11)(31)) ((11)(41)) ((12)(22)) ((12)(32)) ((12)(42))
        ((13)(23)) ((13)(33)) ((13)(43)) ((21)(31)) ((21)(41)) ((22)(32))
        ((22)(42)) ((23)(33)) ((23)(43)) ((31)(41)) ((32)(42)) ((33)(43))

        (11 12 13) (   22 23) (31 32 33) (41 42 43)
              21   ((11)(31)) ((11)(41)) ((12)(22)) ((12)(32)) ((12)(42))
        ((13)(23)) ((13)(33)) ((13)(43)) ((  )(31)) ((  )(41)) ((22)(32))
        ((22)(42)) ((23)(33)) ((23)(43)) ((31)(41)) ((32)(42)) ((33)(43))

        (11 12 13) (   22 23) (31 32 33) (41 42 43)
              21   ((11)(31)) ((11)(41)) ((12)(22)) ((12)(32)) ((12)(42))
        ((13)(23)) ((13)(33)) ((13)(43))                       ((22)(32))
        ((22)(42)) ((23)(33)) ((23)(43)) ((31)(41)) ((32)(42)) ((33)(43))
```

Certainly little gain for such a large amount of work. We now see the downside of virtual insertion: although it is excellent for reducing logical tangles, it is horrible for reducing NP-complex problems, potentially growing at N! rather than 2^N. However, recursive set-insertion is complete.

The technique of recursive set-insertion performs an algebraic contingency analysis for each clause. This differs from case analysis, which performs a contingency analysis for each value of a particular variable.

Symmetry

For a form to be both a tautology and intractable, the variable occurrences must be finely balanced, in raw number and in distribution across parens subforms.

Consider the variable count statistics for the intractable problems above. For each tautology, we observe that the variables are a complete cross-product of two sets and that all occur the same number of times at the same depth of nesting.

3-2 Pigeon-hole (tractable)

```
(11 12)(21 22)(31 32)((11)(21))((11)(31))((12)(22))((12)(32))((21)(31))((22)(32))
```

Indices: {1,2,3} {1,2}
Variables: 6, the set product of the two indices
Clauses: 2-atom = 3, 2-natom = 6
Variable occurrences: for all variables, once at depth=1, twice at depth=2

4-3 Pigeon-hole

```
(11 12 13) (21 22 23) (31 32 33) (41 42 43)
((11)(21)) ((11)(31)) ((11)(41)) ((12)(22)) ((12)(32)) ((12)(42))
((13)(23)) ((13)(33)) ((13)(43)) ((21)(31)) ((21)(41)) ((22)(32))
((22)(42)) ((23)(33)) ((23)(43)) ((31)(41)) ((32)(42)) ((33)(43))
```

Indices: {1,2,3,4} {1,2,3}
Variables: 12, the set product of the two indices
Clauses: 3-atom = 4, 2-natom = 18
Variable occurrences: for all variables, once at depth=1, thrice at depth=2

Three-color Tetrahedron

```
(B1 B2)(B1 B3)(B1 B4)(B2 B3)(B2 B4)(B3 B4)
(G1 G2)(G1 G3)(G1 G4)(G2 G3)(G2 G4)(G3 G4)
(R1 R2)(R1 R3)(R1 R4)(R2 R3)(R2 R4)(R3 R4)
(B1 G1)(B1 R1)(G1 R1)((B1)(G1)(R1))
(B2 G2)(B2 R2)(G2 R2)((B2)(G2)(R2))
(B3 G3)(B3 R3)(G3 R3)((B3)(G3)(R3))
(B4 G4)(B4 R4)(G4 R4)((B4)(G4)(R4))
```

Indices: {B,G,R} {1,2,3,4}
Variables: 12, the set product of the two indices
Clauses: 2-atom = 30, 3-natom = 4
Variable occurrences: for all variables, five times at depth=1, once at depth=2

Three-color Tetrahedron, Recursive Subproblem (tractable)

```
(B2 B3)(B2 B4)(B3 B4)(G2 G3)(G2 G4)(G3 G4)(B2 G2)(B3 G3)(B4 G4)((B2)(G2))((B3)(G3))((B4)(G4))
```

Indices: {B,G,R} {2,3,4}
Variables: 9, the set product of the two indices
Clauses: 2-atom = 6, 2-natom = 3
Variable occurrences: balanced over all variables
twice at depth=1, once at depth=2

It is not known whether or not there is a polynomial symmetry identification algorithm which can reliably identify intractable tautologies.

Non-Symmetry

During set-insertion reduction, we identified subproblems in the context of an insertion. Essentially, the insertion context removes structure, and in the process creates non-symmetrical tautologies. Three non-symmetrical examples follow:

4-3 Pigeon-hole, Recursive Subproblem

```
31 41 (11 12 13)(21 22 23)(32 33)(42 43)((12)(22))((12)(32))((12)(42))((13)(23))
((13)(33))((13)(43))((22)(32))((22)(42))((23)(33))((23)(43))((32)(42))((33)(43))
```

Context: ((11)(21) ^...^)
Indices: {1,2,3,4} {2,3}
Variables: 12, the set product of the two indices
Clauses: 3-atom = 2, 2-atom = 2, 2-natom = 12
Variable occurrences: for all variables without 1 as the second index,
once at depth=1, thrice at depth=2
11, 21 each occurring once at depth 1,
31, 41 each occurring once at depth 0

4-3 Pigeon-hole, Partially Reduced

```
(11 12 13)(22 23)(31 32 33)(41 42 43) 21
((11)(31))((11)(41))((12)(22))((12)(32))((12)(42))((13)(23))((13)(33))((13)(43))
((22)(32))((22)(42))((23)(33))((23)(43))((31)(41))((32)(42))((33)(43))
```

Indices: {1,2,3,4} {1,2,3}
Variables: 12, the set product of the two indices
Clauses: 3-atom = 2, 2-atom = 1, 2-natom = 15

Variable occurrences: for all variables without 1 as the second index,
once at depth=1, thrice at depth=2
11, 31, 41 once at depth=1, twice at depth=2
21 once at depth=0

These partial results indicate that complex tautologies do not necessarily have to be completely symmetrical. The 4-3 pigeon-hole recursive subproblem is balanced when both the context (11)(21) and the inactive atoms 31 41 are taken into account. The partially reduced problem, however, is both non-symmetric and intractable.

Near Symmetry

For a final illustration, here is a nearly symmetrical tautology which is difficult to reduce.

```
(or (and b d) (and b e) (and c d) (and c e) (and a b g)
    (and a c g) (and a d f) (and a e f) (and a f g)
    (nor a b c) (nor b c f) (nor a d e) (nor d e g))

(b d)(b e)(c d)(c e)(a b g)(a c g)(a d f)(a e f)(a f g)          transcribe
((a)(b)(c))((a)(d)(e))((b)(c)(f))((d)(e)(f))
```

Indices: {a,b,c,d,e,f,g}
Variables: 7
Clauses: 2-atom = 4, 3-atom = 5, 3-natom = 4
Variable occurrences: all variables twice at depth=2
variables {b,c,d,e} twice in 2-atom clause, once in 3-atom
variables {e,f} thrice in 3-atom
variable {a}, five times in 3-atom

We cannot call upon symmetry arguments for this problem, so set insertion into every clause is inescapable. For simplification, we might note that all 2-atom clauses fail to reduce since each pair of atoms causes every 3-natom clause to Occlude. With no remaining natoms, reduction will not proceed.

We continue by examining set insertion into the three-natom clauses.

```
(a b g^(b d)(b e)(c d)(c e)(a c g)(a d f)(a e f)(a f g)((a)(b)(c))((a)(d)(e))((b)(c)(f))((d)(e)(f))^)
(a b g^(  d)(  e)(c d)(c e)(  c g)(  d f)(  e f)(  f  )(( )( )(c))(( )(d)(e))(( )(c)(f))((d)(e)(f))^)
(a b g^(  d)(  e)(c d)(c e)(  c g)(  d f)(  e f)(  f  )(( )( )(c))(( )(d)(e))(( )(c)(f))(         )^)

(a c g^(b d)(b e)(c d)(c e)(a b g)(a d f)(a e f)(a f g)((a)(b)(c))((a)(d)(e))((b)(c)(f))((d)(e)(f))^)
(a c g^(b d)(b e)(  d)(  e)(  b  )(  d f)(  e f)(  f  )(( )(b)( ))(( )(d)(e))((b)( )(f))((d)(e)(f))^)
(a c g^(b d)(b e)(  d)(  e)(  b  )(  d f)(  e f)(  f  )(( )(b)( ))(( )(d)(e))((b)( )(f))(         )^)

(a d f^(b d)(b e)(c d)(c e)(a b g)(a c g)(a e f)(a f g)((a)(b)(c))((a)(d)(e))((b)(c)(f))((d)(e)(f))^)
(a d f^(b  )(b e)(c  )(c e)(  b g)(  c g)(  e  )(    g)(( )(b)(c))((a)( )(e))((b)(c)( ))((d)(e)( ))^)
(a d f^(b  )     (c  )                  (  e  )(    g)                                          ^)
(a e f^(b d)(b e)(c d)(c e)(a b g)(a c g)(a d f)(a f g)((a)(b)(c))((a)(d)(e))((b)(c)(f))((d)(e)(f))^)
(a e f^(b d)(b  )(c d)(c  )(  b g)(  c g)(  d  )(    g)(( )(b)(c))((a)(d)( ))((b)(c)( ))((d)( )( ))^)
(a e f^     (b  )     (c  )             (  d  )(    g)                                          ^)

(a f g^(b d)(b e)(c d)(c e)(a b g)(a c g)(a d f)(a e f)((a)(b)(c))((a)(d)(e))((b)(c)(f))((d)(e)(f))^)
(a f g^(b d)(b e)(c d)(c e)(  b  )(  c  )(  d  )(  e  )(( )(b)(c))(( )(d)(e))((b)(c)( ))((d)(e)( ))^)
(a f g^                   (  b  )(  c  )(  d  )(  e  )                                          ^)
```

Two of the five 3-atom clauses vanish. We continue the 3-natom clause insertions without them.

```
((a)(b)(c) ^(b d)(b e)(c d)(c e)(a d f)(a e f)(a f g)((a)(d)(e))((b)(c)(f))((d)(e)(f))^)
((a)(b)(c) ^(b d)(b e)(c d)(c e)(a d f)(a e f)(a f g)(   (d)(e))(      (f))((d)(e)(f))^)
((a)(b)(c) ^(b d)(b e)(c d)(c e)(a d f)(a e f)(a f g)(   (d)(e))        f  ((d)(e)(f))^)
((a)(b)(c) ^(b d)(b e)(c d)(c e)(a d  )(a e  )(a   g)(   (d)(e))        f  ((d)(e)( ))^)
((a)(b)(c) ^(b d)(b e)(c d)(c e)(a d  )(a e  )(a   g)(   (d)(e))        f             ^)
```

Since the natom-level insertion has failed, we continue by an insertion into each natom:

```
((a)(b)(c  ^(b d)(b e)(c d)(c e)(a d  )(a e  )(a   g)(   (d)(e))        f             ^))
((a)(b)(c  ^(b d)(b e)(  d)(  e)(a d  )(a e  )(a   g)(   (d)(e))        f             ^))
((a)(b)(c  ^(b d)(b e)(  d)(  e)(a d  )(a e  )(a   g)(         )        f             ^))
((a)(b)                                                                               )
```

We have succeeded in eliminating one natom; the same insertion can be attempted in the other natoms. We continue, using sequentially accumulated results:

```
((a)(b     ^(b d)(b e)(c d)(c e)(a d  )(a e  )(a   g)(   (d)(e))        f             ^))
((a)(b     ^(  d)(  e)(c d)(c e)(a d  )(a e  )(a   g)(   (d)(e))        f             ^))
((a)(b     ^(  d)(  e)(c d)(c e)(a d  )(a e  )(a   g)(         )        f             ^))
((a)                                                                                  )

((a        ^(b d)(b e)(c d)(c e)(a d  )(a e  )(a   g)(   (d)(e))        f             ^))
((a        ^(b d)(b e)(c d)(c e)(  d  )(  e  )(a   g)(   (d)(e))        f             ^))
((a        ^(b d)(b e)(c d)(c e)(  d  )(  e  )(a   g)(         )        f             ^))
(                                                                                     )
```

This is the desired result, a mark in place of a clause. All other clauses are dominated, and the proof is complete.

CONCLUSION

We have presented a polynomial set-insertion algorithm, based on the BL axiom of Pervasion, that pragmatically discriminates between tractable and intractable tautologies. The algorithm is not complete, which is the basis of its discriminatory capability. When extended, recursive set-insertion is complete, however, the extension introduces an exponential effort which is significantly worse than case analysis. The polynomial algorithm is sufficient for identifying tautologies constructed using logical inference and replacement rules and any embeddings of these rules within themselves.

Table I: The Map from Logic to Boundary Logic

logic	*boundary*	*comments*
False	<void>	no representation
True	( )	the empty boundary
A	A	objects are labeled by tokens
not A	(A)	negation: inside a boundary
A or B	A B	disjunction: sharing space
A and B	((A)(B))	conjunction: bounded bounds
if A then B	(A) B	implication: separation by a boundary
A iff B	(A B)((A)(B))	the source of variable replication

Table IIa: Axiomatic Bases for Conventional Logic

Conventional Implicational Basis:

p -> (q -> p) *isTrue*

((p -> False) -> False) -> p *isTrue*

(p -> (q -> r)) -> ((p -> q) -> (p -> r)) *isTrue*

Transcribing into boundary logic:

(p) (q) p = ( )

(((p))) p = ( )

((p) (q) r) ((p) q) (p) r = ( )

Huntington's Basis for Boolean Algebra:

Commutativity	*a+b = b+a*	*a*b = b*a*
Identity	*a+**0** = a*	*a***1** = a*
Complement	*a+a' = **1***	*a*a' = **0***
Distribution	*a+(b*c) = (a+b)*(a+c)*	*a*(b+c) = (a*b)+(a*c)*

Transcribing into boundary logic:

a b = b a	((a)(b)) = ((b)(a))
a = a	((a)( )) = a
a (a) = ( )	((a)((a))) = <void>
a ((b)(c)) = ((a b)(a c))	((a)(b c)) = ((a)(b))((a)(c))

Table IIb: Axiomatic Bases for Boundary Logic

Spencer-Brown's Axioms in Laws of Form

Position	(A (A)) = <void>
Transposition	A ((B)(C)) = ((A B)(A C))

Computational Basis for Boundary Logic:

Occlusion	(( ) A) = <void>
Involution	((A)) = A
Pervasion	A {B A} = A {B}

Recursive Basis for Boundary Logic:

Base case:	(( ) A) = <void>	*Occlusion*
Inductive case:	A {B A} = A {B}	*Pervasion*

Kauffman's Single Axiom for Boundary Logic:

Extension	(A B)(A (B)) = (A)

Table III: Proving the Rules of Inference using Boundary Logic

Modus Ponens	p and (p->q) |= q	(((p)((p) q))) q (p)((p) q) q (p)( ) q
Modus Tollens	~q and (p->q) |= ~p	((((q))((p) q))) (p) q ((p) q) (p) q ( ) (p)
Hypothetical Syllogism	(p->q) and (q->r) |= p->r	((((p) q)((q) r))) (p) r ((p) q)((q) r) (p) r (q)((q)) (p) r (q)( ) (p) r
Disjunctive Syllogism	(pVq) and ~p |= q (pVq) and ~q |= p	(((p q)((q)))) p (p q) q p ( ) q p
Dilemma	(p->q) and (r->s) and (pVr) |= qVs	((((p) q)((r) s)(p r))) q s ((p) q)((r) s)(p r) q s ((p))((r))(p r) q s p r (p r) q s p r ( ) q s
Simplification	p&q |= p p&q |= q	(((p)(q))) q (p)(q) q (p)( ) q
Conjunction	p and q |= p&q	(((p)(q))) ((p)(q)) (p)(q) ((p)(q)) (p)(q) ( )
Addition	p |= pVq q |= pVq	(q) p q ( ) p q
Cases	(p->q) and (~p->q) |= q	((((p) q)(((p)) q))) q ((p) q)(p q) q ((p))(p) q ( )(p) q
Inconsistency	p and ~p |= q	(((p)((p)))) q (p) p q ( ) p q

Table IV: Proving the Logic Replacement Rules

Excluded Middle	pV~p =||= T	(p) p = ( ) ( ) p = ( )
Contradiction	p&~p =||= F	((p)((p))) = <void> ((p)( )) = <void> (( )) = <void>
Domination	pVT =||= T pVF =||= p p&F =||= F p&F =||= p	p ( ) = ( ) p = p ((p)( )) = <void> (( )) = <void> ((p)(( ))) = p ((p)) = p p = p
Double Negation	p =||= ~~p	p = ((p)) p = p
Duplication	p =||= pVp p =||= p&p	p = p p p = p p = ((p)(p)) p = ((p)) p = p
Commutation	pVq =||= qVp p&q =||= q&p	p q = q p ((p)(q)) = ((q)(p))
Association	(pVq)Vr =||= pV(qVr) (p&q)&r =||= p&(q&r)	p q r = p q r ((((p)(q)))(r)) = ((p)(((q)(r)))) ((p)(q) (r)) = ((p) (q)(r))
Contraposition	p->q =||= ~q->~p	(p) q = ((q)) (p) (p) q = q (p)
DeMorgan's	~(pVq) =||= ~p&~q ~(p&q) =||= ~pV~q	(p q) = (((p))((q))) (p q) = (p q) (((p)(q))) = (p)(q) (p)(q) = (p)(q)
Conditional Exchange	p->q =||= ~pVq	(p) q = (p) q
Exportation	(p&q)->r =||= p->(q->r)	(((p)(q))) r = (p)(q) r (p)(q) r = (p)(q) r
Biconditional Exchange	p=q =||= (p->q)&(q->p) p=q =||= ~(pVq)V(p&q)	p=q = (((p) q)((q) p)) p=q = (p q) ((q)(p))

Table V: Non-trivial Proofs of Logic Replacement Rules

Absorption

pV(p&q) =||= p

```
p ((p)(q)) = p
p (( )(q)) = p
p (( )   ) = p
p          = p
```

p&(pVq) =||= p

```
     ((p)(p q)) = p
((p)(^(p)^ p q)) = p
((p)(^( )^ p q)) = p
((p)           ) = p
  p              = p
```

Distribution

p&(qVr) =||= (p&q)V(p&r)

```
              ((p)(q r)) = ((p)(q))((p)(r))
((p                    )(     q       r  ))
((p                    )((   (q))(   (r))))
((p                    )(((p)(q))((p)(r))))
((p^(((p)(q))((p)(r)))^)(((p)(q))((p)(r))))
((p^((( )(q))(( )(r)))^)(((p)(q))((p)(r))))
((p^(                )^)(((p)(q))((p)(r))))
(                      (((p)(q))((p)(r))))
                        ((p)(q))((p)(r))
```

pV(q&r) =||= (pVq)&(pVr)

```
     p ((q)(r)) = ((p q)(p r))
  p               ((p q)(p r))
((p)             )((p q)(p r))
((p)^((p q)(p r))^)((p q)(p r))
((p)^(          )^)((p q)(p r))
                    ((p q)(p r))
```

Biconditional Identity

p=p =||= T

```
((p) p)((p) p)) = ( )
(( ) p)(( ) p)) = ( )
(             ) = ( )
```

Biconditional Negation

p=~q =||= ~(p=q)

```
 (((p)(q))(((q)) p)) = ((((p) q)((q) p)))
 (((p)(q))(  q   p)) =   ((p) q)((q) p)
(((p      )(q      ))(                 q p))
(((p (q p))(q (q p)))(                 q p))
(((p (q  ))(q (  p)))(                 q p))
(((p (q  ))(q (  p)))(((p (q))(q (p))) q p))
(((p (q  ))(q (  p)))(((  ( ))(  ( ))) q p))
(((p (q  ))(q (  p)))((              ) q p))
(((p (q  ))(q (  p)))                    )
  (p (q  ))(q (  p))
```