Modeling with Logic

The Modeling Hierarchy

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

Mathematical Structures

propositional calculus (Boolean algebra)
- truth symbols
- propositional symbols (binary variables)
- connectives (and, or, not)
- interpretations

predicate calculus
- truth symbols
- constant symbols
- variable symbols
- function symbols
- predicate symbols (relations)
- quantifiers

equality and orderings
non-negative *integers*
data *structures*

strings	trees	lists	sets
bags (multi-sets)		tuples	graphs

Formal Modeling

A *formal system* (a mathematical system) consists of

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

Formal = Atoms + Forms + Transforms + Axioms

Aristotle

Aristotle classified declarative language using three polar categories

single vs compound (Socrates is happy. vs Man is happy.)
universal vs particular
affirm vs deny

The latter two categories form the ***Square of Opposition***

	AFFIRM	*DENY*
UNIVERSAL	A Every _ is _ . (Everyone is happy.)	E No _ is _ . (No one is happy.)
PARTICULAR	I Some _ is _ . (Someone is happy.)	O Some _ is not _ . (Someone is not happy.)

The Syllogism

The Syllogism according to Aristotle:
"discussion in which, when things are posited, other things necessarily follow."

All men are mortal.
Socrates is a man.
Thus, Socrates is mortal.

This non-Aristotelian form of logic was developed into
the first ever ***Axiomatic System*** with variables.

Crisis in the Twentieth Century

Oh No! There is ***no consistency in mathematics***, there are paradoxes in every system.

Logicism (Russell)
Mathematics is identical to logic. (We'll patch the holes.)

Intuitionism (Brouwer)
Mathematics presupposes concepts. Concepts rest on natural numbers.
(We'll construct what is known, and not admit infinity.)

Formalism (Hilbert)
Mathematics is a set of syntactic transformations. (We'll refuse to interpret it.)

Predicate Calculus

A g***eneral purpose*** language for describing objects, facts, and transformations for particular domains. Also called ***First Order Logic***. It consists of

connective logic	{and, or, if, not, iff} inference, proof
object domains	{<unique atoms>}
quantification	{all x, exists x}
predicates	classes and properties
relations	True associations between objects
functions	indirect names, maps from one object to another

Proof Techniques

Mechanisms of Proof

Truth tables	exhaustive analysis of all possible cases
Natural deduction	using language and intuition to figure out truths
Resolution	a computational algorithm for deduction
Boundary logic	a new form of spatial representation and parallel deduction

Generic Applications of Proof Theory

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.

Motivation for the Hard Problem

The essential problem is that ***proof is difficult***. How logic is represented and how it is transformed determine the ease of (all) computation.

- Multiplicity ought not be posited without necessity. (Occam)

- Programming = data structure + algorithm (Wirth)

- Proof = definition + axiom (Lakatos)

- A good representation for a problem permits simple transformations to a solution
  however
  any given representation blinds us to other ways of thinking. (Winograd)

Truth Table Analysis

Examining all possibilities is exponential in number of cases: there are 2^n cases to evaluate for n variables even in the simplest case of propositional logic without functions or relations. However, lookup tables are a brute force algorithm that is easy to understand and to implement. The technique is to list all possible combinations of values for each variable, and use simple definitions of the logical connectives to evaluate compound sub-expressions.

Example: `if (P and Q) then (R = (not S))`

P	Q	R	S	(not S)	(P and Q)	(R = (not S))	(if P&Q then R=~S)
T	T	T	T	F	T	F	F
T	T	T	F	T	T	T	T
T	T	F	T	F	T	T	T
T	T	F	F	T	T	F	F
T	F	T	T	F	F	F	T
T	F	T	F	T	F	T	T
T	F	F	T	F	F	T	T
T	F	F	F	T	F	F	T
F	T	T	T	F	F	F	T
F	T	T	F	T	F	T	T
F	T	F	T	F	F	T	T
F	T	F	F	T	F	F	T
F	F	T	T	F	F	F	T
F	F	T	F	T	F	T	T
F	F	F	T	F	F	T	T
F	F	F	F	T	F	F	T

The Rules of Natural Deduction

Natural deduction evolved from natural language and from human intuition, so it is relatively easy to understand. It is very difficult to find the right rules to apply at the right time (exponential in difficulty of use). Recall that humankind has had an extremely difficult time understanding logic, and logic itself is still undergoing extreme revision. The name of the primary rule of natural deduction (*modus ponens*) is still in Latin. "|=" means "logically implies" while "**->**" is simply a symbol referring to a specific truth table. The same subtle difference exists between "**and**" and "**&**".

Modus Ponens:	A	and	A -> B	|=	B
Modus Tollens:	~B	and	A -> B	|=	~A
Double negation:	A			|=	~~A
	~~A			|=	A
Conjunction:	A	and	B	|=	A & B
Simplification:	A & B			|=	A
	A & B			|=	B
Addition:	A			|=	A v B

Natural Deduction Proof Techniques

Modus Ponens:	`A and A |- B`	|=	`B`
Modus Tollens:	`~B and A |- B`	|=	`~A`
Conditional proof:	`A |- B`	|=	`A -> B`
Dilemma:	`(A or B) and (A |- C) and (B |- C)`	|=	`C`
Contradiction:	`(A |- B) and ~B`	|=	`~A`
Cases:	`(A is True |- B) and (A is False |- B)`	|=	`B`

Note that "l-" is a sequence of formal steps, while "l=" is assurance of logical truth.

Generalization of Deduction

The rules of inference, or natural deduction, apply at three different ***levels of abstraction***: individual propositions, individual sentences, and collections of sentences.
Modus Ponens serves as an example.

Atoms: (p and (p implies q)) implies q

Sentences: (A and (A implies B)) implies B

Collections of sentences: ({A,B...} and ({A,B...} implies {C,D...})) implies {C,D...}

Deductive Steps

There are three separate concepts of ***proof step*** (written above as "implies") which have been shown to be equivalent: material implication, logical implication, and entailment.

Material implication: p -> q

p	**q**	**(p -> q)**
0	0	1
0	1	1
1	0	0
1	1	1

Material implication is defined by the *Truth Table* of values.
Notice that the second row, "`(if False then True) is True`", does not make sense in language structures, it is True by definition.

Logical Implication: p |= q

"if (p is logically True) then (q is logically True)"

Logical implication is defined by *common sense* and by the rules of deduction.

Formal Proof: p |- q

"if (p is True) then a sequence of implications shows (q is True)"

Formal proof is defined by taking *logical implication steps* from p to q

Natural Deduction Example

In English:

Premise 1:	If he is lying, then (if we can't find the gun, then he'll get away).
Premise 2:	If he gets away, then (if he is drunk or not careful, then we can find the gun).
Premise 3:	It is not the case that (if he has a car, then we can find the gun).
Conclusion:	It is not the case that he is both lying and drunk.

Encode the propositions as letters:

L = he is lying
G = we can find the gun
A = he will get away
D = he is drunk
C = he is careful
H = he has a car

Premise 1:	`If L then (if (not G) A)`
Premise 2:	`If A then (if (D or not C) then G)`
Premise 3:	`Not (if H then G)`
Conclusion:	`Not (L and D)`

Encode the logical connectives:

```
P1:   L -> (~G -> A)
P2:   A -> ((D v ~C) -> G)
P3:   ~(H -> G)
C:    ~(L & D)
```

Figure out a good proof strategy. This step is not algorithmic, and is the source of difficulty in natural deduction approaches. Here the Contradiction strategy works:

1.	`(L & D)`	assume the negated conclusion, plan to show a contradiction
2.	`L`	simplification of 1
3.	`D`	simplification of 1
4.	`~G -> A`	modus ponens with 2 and P1
5.	`~(~H v G)`	rewrite P3 with conditional exchange: `X -> Y = ~X v Y`

6.	`~(~H v ~~G)`	double negation of part of 5
7.	`H & ~G`	rewrite 6 with DeMorgan: `~(~X v ~Y) = X & Y`
8.	`~G`	simplification of 8
9.	`A`	modus ponens with 8 and 4
10.	`((D v ~C) -> G)`	modus ponens with 9 and P2
11.	`(D v ~C)`	addition of `~C` to 3
12.	`G`	modus ponens with 11 and 10
13.	`G & ~G`	conjunction of 8 and 12
14.	`~(L & D)`	contradiction 1-13

Steps from 1 to 13 have created a contradiction: `G & ~G = False`, therefore the assumption on line 1 is false. But that assumption is the negation of the conclusion. Therefore the negation of the negation of the conclusion is True. That is, the conclusion is True.

Variable Renaming

Variables are used in algorithmic deduction as arguments to functions and relations. In this case, it is necessary to assure that each variable name is ***unique***. This is in contrast with the standard (pre-computational) conventions of logic and algebra which use the same variable names in different rules. The convention existed because quantification "protected" the variable names by scoping rules, just as local variable declarations protect name clashes in programming.

Conventional:

```
R1.   if P(x) then Q(x)
R2.   if R(x,y) and R(y,z) then R(x,z)
R3.   if P(x) and R(x,y) then P(y)
```

Computational: assign all variables unique names.

```
R1.   if P(_1_) then Q(_1_)
R2.   if R(_2_, _3_) and R(_3_, _4_) then R(_2_, _4_)
R3.   if P(_5_) and R(_5_, _6_) then P(_6_)
```

Unification

Facts do not contain variables, functions and relations can contain variables since they are intended to be abstractions. (The lexicographic arrangement of a form is an arbitrary choice, but it must be consistent within a database for matching to succeed.) Facts are combined with functions and relations through unification, ***a generalization of pattern-matching***. Pattern-matching comes in three varieties:

1. The pattern is a specific ground (a fact with no variables). A match is exact, typographical character by character. Note that no conclusions follow if a pattern is not in the database.

```
(john is-parent-of sue)  =match=  (john is-parent-of sue)
```

2. The pattern contains variables which match grounds in the database. In this case, more than one match is possible. The type of pattern variable can be used to control matching. Below, the bindings are _1_ = {john, betty}.

```
(_1_ is-parent-of sue)  =match=  (john is-parent-of sue)
                        =match=  (betty is-parent-of sue)
```

3. Both the pattern and the data being matched contain variables. This matching procedure is called ***unification***. Below, the bindings are _1_ = {_2_} and _3_ = {sue}. The result is the further query: (_2_ is-ancestor-of sue)

```
(_1_ is-parent-of sue)  =unify=
       if (_2_ is-parent-of _3_) then (_2_ is-ancestor-of _3_)
```

In matching, only variables from one side of the equation can be matched. In unification, variables on either side can be matched. Examples:

```
(parents-of _1_ are (father-of tom) (mother-of sue) )  =unify=
                (parents-of sue are (father-of _2_) _3_)
```

bindings: _1_ = { sue }, _2_ = { tom }, _3_ = { Mother(sue) }

```
(_1_ likes _2_) and (sue likes tom)  =unify=
                (tom likes sue) and (not (_3_ likes _4_))
```

bindings: fail. The word "not" does not match.

```
F( G(a, _1_), H(_2_, J(_2_, _1_)) ) =unify= F(_3_, H( K(_3_), J(K(_3_), d)))
```

bindings: _1_ = {d}, _2_ = { K(G(a, d)) }, _3_ = { G(a, d) }

Resolution

In 1965, John Robinson devised a computational approach to proof called ***resolution***. The resolution principle uses the two possible cases of one variable. If the fact that a thing is True leads to one conclusion, and the fact that it is False leads to another conclusion, then in any case either the first or the second conclusion is True.

```
((if P then Q) and (if (not P) then R))  implies (Q or R)
```

As a deductive rule, resolution can be stated in increasing general forms:

```
         P          and    ~P or False       |=   False
         P          and      ~P or Q         |=     Q
      P or Q        and      ~P or Q         |=     Q
      P or Q        and      ~P or R         |=   Q or R
  (P and U) or Q  and..(~P and V) or R       |=  (U or Q) or (V or R)
```

Resolution proof uses a ***clausal*** data structure consisting of sets of literals in disjunction. A pair of sets, one with a positive occurrence of a variable and one with a negative occurrence, is resolved by forming the union of the two sets, and deleting the resolvent variable.

```
{p, q} union {~p, r}    |=  {q, r}
```

Facts are expressed as a singular set:

```
{p}
```

Rules are converted from implicational form to disjunctive form:

```
p -> q  ==>  ~p or q   ==>   {~p, q}
```

When the resolvent atoms have internal structure (functions and relations), the internal variables are ***unified*** in the course of resolving the atoms.

Resolution Example

Example: Who are Jon's parents?

```
F1.    (Bob is-father-of Jon)              =     {bFj}
F2.    (Sue is-mother-of Jon)              =     {sMj}
R1.    If (_1_ is-father-of _2_)
             then (_1_ is-parent-of _2_)   =     {~1F2, 1P2}
R2.    If (_3_ is-mother-of _4_)
             then (_3_ is-parent-of _4_)   =     {~3M4, 3P4}
Q.     If (_5_ is-parent-of Jon)
             then (answer-is _5_)          =     {~5Pj, A5}
```

```
Resolve  C1.   F1 and R1   =     {bPj}
         C2.   F2 and R2   =     {sPj}
         C3.   R1 and Q    =     {~5Fj, A5}
         C4.   R2 and Q    =     {~5Mj, A5}  rename  {~6Mj, A6}

         C5.   Q and C1    =     {Ab}
         C6.   Q and C2    =     {As}
         C7.   F1 and C3   =     {Ab}
         C8.   F2 and C4   =     {As}
```

The algorithm completes all available resolutions, so that complete resolution over a database usually generates the same answer more than once. Database queries distinguish whether one match, or answer, is requested, or whether all matches are requested.

Quantification

Quantifiers introduce sets into logic, and serve to define the scope of variables in a logical expression.

Universal quantification: `All x. P(x)`

The statement `All x. P(x)` is True exactly when the predicate `P` (or the characteristic function for the set P) is True for all objects in the set U for which `x` is an arbitrary member.

For finite domains U, `All x. P(x)  iff  (x1 and x2 and ... and xn)`

Existential quantification: `Exists x. P(x)`

The statement `Exists x. P(x)` is True exactly when the predicate `P` is True for at least one object in the set U for which `x` is an arbitrary member.

For finite domains U, `Exists x. P(x)  iff  (x1 or x2 or ... or xn)`

Relationships between Quantifiers

All true = none false:	`All x. P(x)  iff  (not (Exists x. (not P(x))))`
All false = none true:	`All x. (not P(x))  iff  (not (Exists x. P(x)))`
Not all true = at least one false:	`(not (All x. P(x)))  iff  Exists x. (not P(x))`
Not all false = at least one true:	`(not (All x. (not P(x))))  iff  Exists x. P(x)`

Algebraic Proof Techniques

Standard Form (Boolean systems only)

```
A = B       iff   ((A -> B) & (B -> A))  isTrue
```

Direct Transformation

```
A = B       iff    A |- B  or  B |- A
```

Mutual Transformation

```
A = B       iff    A |- C  and  B |- C
```

Case Analysis (Z: X => Y means "Y is substituted for X in expression Z")

```
A = B       iff   (A:  E =>  True) = (B: E =>  True)
            and   (A:  E => False) = (B: E => False)
```

Complexity in Computation

Model Theory

Given an object domain and a collection of functions and relations on objects in that domain, a ***model*** of the domain is defined by its facts:

all atoms (atom-names) in the domain are True
all atoms not in the domain are False

Example: Object domain = {Mary, Tom, John}   Relation: {Likes}

all possible atoms:	*all possible models*:	
Likes[Mary, Tom]	1	empty state (no atoms true)
Likes[Mary, John]	6	one Likes atom isTrue
Likes[John, Mary]	15	two Likes atoms are True
Likes[Tom, Mary]	20	three True
Likes[John, Tom]	15	four True
Likes[Tom, John]	6	five True
	1	six True
	64	possible models in total

The set of all possible models is called the ***Herbrand Universe***. In theories with integers or theories with functions, this universe is usually infinite.

The Equivalence of Deduction and Computation

From 1920 to 1970, a central issue for theoretical computer science was whether a computation maintained the meaning of a logical (deductive) process. Fortunately the answer is yes, given that the programming language follows the rules of logic. These are the essential concepts:

P |- Q   Single turnstile: Q is ***computed*** from P

P |= Q   Double turnstile: Q is ***deducible*** from P

Soundness: If P |- Q, then P |= Q

A sound computation always maintains the deductive model. The computation never generates a False model, it never produces an incorrect fact.

Completeness: If P |= Q, then P |- Q

A complete computation never varies from the deductive model. The computation generates all True models. If something can be deduced, the computation will be able to deduce it.

Sound and Complete: P |- Q = P |= Q

The deductive model and the computation represent the same Universe. The computation produces all that can be deduced, and only what can be deduced.

Decidability

Universal:

If it can be computed formally, then it can be computed using First Order Logic (Turing equivalence).

Decidable:

The computational procedure will terminate with a Yes/No result.

Semi-decidable:

The computation might halt, but you don't know when. It may never halt if you ask the wrong kind of question. What we can't do is ask questions which depend on the **failure** to prove something:

No: "Check to see if nothing is wrong"
No: "Prove that this search will fail to find X"

SAT and TAUT

The ***most important theoretical question*** in Computer Science (with the most critical practical applications) is

Can we write an algorithm which runs
in *polynomial space and time* with regard to the number of variables
that shows that an arbitrary expression in Propositional Calculus
has a *non-trivial variable* in it?

SAT: Is a Boolean expression SATisfiable?

That is, is there an assignment of variable values (True or False) which results in different outcomes for the entire expression (again either True or False)?

TAUT: Is a Boolean expression a TAUTology

That is, is every variable in the expression irrelevant? Is the expression always True regardless of the values assigned to the variables in it?

If TAUT or SAT is solvable within polynomial bounds, then **P = NP**

P =?= NP

Almost all complex but common computational problems can be mapped onto the SAT problem. If you can show propositional satisfiability within polynomial bounds, then most computational problems have an efficient solution. In particular, programs and circuits can be verified efficiently.

Polynomial complexity: O[n^some-power]

some polynomials: `n^2        (n+1)(n - 1)        14n^6 + 3n^3 + n + 104`

Non-deterministic polynomial complexity: O[2^n] or worse

some non-polynomials: `2^n        7^(n + 3)        n!        2^(2^n)`

Polynomial algorithms are called ***tractable***: they all run in "reasonable" time. Non-polynomial algorithms are intractable: none of them are efficient enough to be useful for problems of arbitrary size.

Irrelevant Variables

In 1930, Huntington provided the first proof that irrelevant logical structures of arbitrary complexity can be introduced into a logical form. Today, we can easily see how variables can occur in a logical form (or in a computational control structure) without contributing to the meaning of the form.

(A or True) = True

The logical law of Addition, `A |= (A or B)`, permits the inclusion of arbitrary forms through disjunction. In a program, the order of execution of two forms joined by OR is important, because one of the forms may render the computation of the other irrelevant. However, compilers can search for the "True" token in linear time, and simplify the code accordingly. It is no more difficult when the irrelevant forms are hidden in multiple disjunction:

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

(A or ~A) = True

The Law of Excluded Middle is more difficult, since it appears that "`A`" might need to be evaluated at least once. However, if the pattern "`A`" can be located in two different places by pattern-matching, it can be coalesced into one. Code compilers do this so that the subroutine "`A`" is evaluated only once, and the result, rather than the call, is stored in memory. If the coalesced form is in the relation of Excluded Middle, it can be eliminated altogether, without evaluation. The form of "`A`", however, may look different in different places, due to other logical invariants. For example:

```
A and B  =  B and A
```

could make Excluded Middle more complex:

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

This problem is resolved by sorting the form into a standard, or canonical, ordering. Since sorting can be achieved efficiently, this type of logical embedding is not the source of complexity (although it does require more computation to avoid.

X = Y iff (X -> Y) & (Y -> X) = True

In the Boolean domain, an algebraic equivalence can always be converted into a Truth statement in standard form. 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
```

This too can be addressed by sorting the form into a canonical ordering.

Boundary Techniques

Boundary Logic

Boundary logic uses a ***spatial representation*** of the logical connectives. Because boundaries delineate both objects and processes, boundary forms can be evaluated using either an algebraic (match and substitute) process or a functional (input converted to output) process.

Representation of logic and proof in spatial boundaries is new, and quite unfamiliar. Boundary logic is not based on language or on typographical strings, nor is it based on sequential steps. Boundary techniques are inherently ***parallel and positional***. The meaning, or interpretation, of a boundary form depends on where the observer is situated. From the outside, boundaries are objects. From the inside, you cross a boundary to get to the outside; boundaries then are processes. This dramatically different approach (that is, permitting the observer to be an operator in the system) does not change the logical consequences or the deductive validity of a logical process.

Spatial representations have built-in associativity and commutativity. The base case is no representation at all, that is, ***the void has meaning*** in boundary logic. Logical expressions are simplified by ***erasure of irrelevancies*** rather than by accumulation of facts.

Boundary Logic Representation

logic	*boundary*	*comments*
`False`	`<void>`	no representation. Note: `(()) = <void>`
`True`	`(   )`	the empty boundary
`A`	`A`	objects are labeled by names
`not A`	`(A)`	
`A or B`	`A   B`	Disjunction is sharing the same space
`A and B`	`((A)(B))`	
`if A then B`	`(A) B`	Implication is separating by a boundary
`A iff B`	`(A B)((A)(B))`	

In the above map from conventional logic to boundaries, the many textual forms of logical connectives condense into one boundary form. Note that the parens, `(  )`, is a linear, or one-dimensional, representation of a boundary. Circles and spheres are expressions of boundaries in higher dimensional representations.

Multiple Readings of the Same Form

A simpler notation expresses in one form the many different forms of a more complex notation. For example:

```
            ( (A) (B) )       A and B
                              (not ((not A) or (not B)))
                              (not (A implies (not B)))
                              ((not A) or (not B)) implies False
```

A proof of DeMorgan's Law:

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

Transcribe: `((A)(B)) = ((A)(B))` equal by identity

Boundary Logic Algebraic Process

The transformation axioms of boundary logic:

Dominion (the halting condition, when to stop)

```
( ) A  =  ( )
```

Involution (double negation, how to remove excess boundaries)

```
((A))  =  A
```

Pervasion (how to remove excess occurrences of variables)

```
A (A B)  =  A (B)
```

Each axiom suggests a definite reduction strategy: ***erasing irrelevant structure*** to convert the left side of the equation to the right side.

A Boundary Logic Example: Proving Modus Ponens

Modus Ponens: `(A and (A implies B)) implies B`

Transcribe: `( ((A) ((A) B)) )  B`

Simplify (prove):

```
(  ( ( A ) ( (A) B ))  )  B

   ( A ) ( (A) B )      B         involution
   ( A ) ( (A)   )      B         pervasion of B
   ( A )    A           B         involution
   (   )    A           B         pervasion of A
   (   )                          dominion
```

The remaining form is transcribed back into logic as True.

Truth Table Example in Boundary Logic

Example: `if (P and Q) then (R = (not S))`

Encode as boundaries:

```
(P  and  Q)         ((P) (Q))
(R = (not S))       (R (S)) ((R)((S)))  =  (R (S)) ((R) S)

if...then...        (((P) (Q))) (R (S)) ((R) S) = (P) (Q) (R (S)) ((R) S)
```

The expression is True whenever Dominion applies

```
When P is False, it is erased:  ( ) (Q) (R (S)) ((R) S)  =  ( )   dominion
When Q is False:                 (P) ( ) (R (S)) ((R) S)  =  ( )   dominion
```

Note that the form `(X (Y)) (Y (X))` is True when `X` is not the same as `Y`, so the expression is True when R is False and S is True. Substituting:

```
(P) (Q) ( (( ))) (( ) ( ))  =  (P) (Q)  ( )        =  ( )
```

and when R is True and S is False

```
(P) (Q) (( ) ( )) ( (( )))  =  (P) (Q)        ( )  =  ( )
```

These four cases identify all the True forms of the expression.

Conversely, the expression is False only when everything vanishes, that is, when

(P is True) and (Q is True) and ((R and S are both True) or (R and S are both False))

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

Natural Deduction Example in Boundary Logic

Premise 1: `If  L  then  (if  (not  G)  A)`
Premise 2: `If  A  then  (if  (D  or  not  C) then G)`
Premise 3: `Not (if  H  then  G)`
Conclusion: `Not (L and D)`

Encode the logical connectives as boundaries, and simplify:

```
P1:        (L) ((G)) A    =  (L) G A          involution
P2:        (A) (D (C)) G
P3:        ((H) G)
C:         (((L)(D)))     =  (L)(D)           involution
```

Join all premises and conclusions into one form, using the logical structure:

```
            (P1 and P2 and P3) -> C
```

The proof structure of "conjunction of premises imply the conclusion" as boundaries:

```
            ( ((P1) (P2) (P3)) ) C  =  (P1) (P2) (P3) C       involution
```

Substituting the forms of the premises and conclusion, and reducing:

```
    ( (L) G A )  ( (A) (D (C)) G )  ( ((H) G) )  (L) (D)
    ( (L) G A )  ( (A) (D (C)) G )     (H) G     (L) (D)      involution
    ( (L)   A )  ( (A) (D (C))   )     (H) G     (L) (D)      pervasion of G
    (       A )  ( (A) (D (C))   )     (H) G     (L) (D)      pervasion of (L)
    (       A )  (     (D (C))   )     (H) G     (L) (D)      pervasion of (A)
    (       A )         D (C)          (H) G     (L) (D)      involution
    (       A )         D (C)          (H) G     (L) ( )      pervasion of D
                                                     ( )      dominion
```

Interpret the final form: `( ) = True`

Boundary Quantification

```
     All x. P(x)        (x) Px            x implies Px  isTrue

     Exists x. P(x)     ((x)(Px))         x and Px  isTrue
```

Quantifier relations:	Standard form	Boundary form
	`All x. P(x)  iff  (not (Exists x. (not P(x))))`	`(x) Px = (((x)((Px))))`
	`All x. (not P(x))  iff  (not (Exists x. P(x)))`	`(x)(Px) = (((x)(Px)))`
	`(not (All x. P(x)))  iff  Exists x. (not P(x))`	`((x) Px) = ((x)((Px)))`
	`(not (All x. (not P(x))))  iff  Exists x. P(x)`	`((x)(Px)) = ((x)(Px))`