Logic and Computation

From Conceptualization to Formalization

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

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

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

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

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

3. Use different labels to **name** different *abstract* things:

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

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

In summary, we convert from physical to virtual, ignore the physical aspects of reality, manipulate the virtual (or digital) aspects using the rules of virtuality, and then return to physical reality with new knowledge. The rules of virtuality are listed in the handout *The Canons of Formal Symbol Systems*.

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

Computational hardware is engineered (constructed) to behave as if it were a mathematical system called *Timed Boolean Logic*. “Timed” simply means that some parts of the physical hardware are used more than once to do the Boolean logic task for which they were constructed.

Formal Simplicity

The simplest interesting formal domain is **Propositional Calculus**, also known as **Boolean Logic**, and also known as **Boolean Algebra**.

In Boolean algebra, the labels for real world objects are sentences in human language which can be evaluated as either True or False. These sentences are composed of atoms (propositions) and logical connectives. The atoms are labels for things which exist and are in some way actual, or real, or True. The logical connectives are those parts of language which do *not* refer to something in physical reality.

More generally, Boolean algebra includes all of *decision theory*, which phrases a problem space as a sequence of binary decisions. (Note that more complex decisions can usually be phrased as a collection of binary decisions.)

The truth of a sentence, the direction of a decision, and the voltage in a digital circuit wire are all binary properties. Every object label in a Boolean problem has the *binary property*, either 0 or 1, and that is the only property of interest. Logic is the set of rules which maintains the invariance of the value of binary properties.

Labeling Reality

Let {a, b, c, ...} be simple Boolean variables identifying a set of decision events. Encode the value of each decision (yes/no, true/false, act/pass) as 1/0.

Combinations of Boolean operators and Boolean variables are called *sentences* or *expressions*.

Let {A, B, C, ...} be names of Boolean expressions.

Valid Boolean expressions are defined recursively,

a	simple variables are expressions
not A	negated expressions are expressions
A or B	expressions joined by disjunction are expressions

Since all other Boolean operators can be expressed in terms of **not** and **or**, they are all included in the above definition. This definition means that Boolean expressions are composable and decomposable; the recursive rules provide Constructors and Accessors.

Symbolic Complexity

The truth table (or Boolean property table) of a Boolean operator specifies how that operator combines the Boolean properties, or the values, joined by the operator. But here things start to get complex. Mainly, we can form a huge variety of expressions, and to find the Boolean value of any expression, we must examine a table with a size exponential in the number of variables (2^n entries). So a logic circuit with 200 input variables requires an impractical 2^200 rows in the truth table.

So all decision problems which require a large number of decisions become intractably inconvenient to examine in the whole. Boolean algebra provides *transformation rules* which hold for all expressions, regardless of which Boolean value an expression has. For example,

not (not A) = A

is valid regardless of the expression A.

Axioms and Theorems

Transformation rules can be decomposed into axioms and theorems. The axioms are a minimal set of rules which preserve the Boolean property and serve as a basis for all other possible theorems. All the axioms of an axiomatic formal system can be compressed into one single, usually complex axiom. For Boolean logic, the simplest single axiom which supports all other Boolean transformations is the Kauffman/Flagg formalization of the Robbins problem:

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

Similarly, the Resolution rule is a single rule which supports all Boolean transformation. The many rules of Natural Deduction provide another, more complex set of axioms. One particularly simple set of supporting axioms (Bricken's formalization of Boundary Logic) is the following:

A or True = True	DOMINION
not (not A) = A	INVOLUTION
A or (not (A or B)) = A or (not B)	PERVASION

This set is particularly useful for pattern-matching and automated deduction, since the patterns to be matched are relatively simple, and the right-hand-side of each equation is simpler than the left-hand-side solely through erasing something from the right-hand-side.

Proof

Boolean algebra axioms and theorems (valid transformations) provide a way to explore decision spaces without making the actual decisions. This is called *logical or algebraic proof*. However, the situation remains complex because now we must select which theorem to apply and where to apply it. Although the search space is more abstract, it is still intractable and inconvenient. Although Boolean algebra abstracts the physical properties of decisions, it is still a *real world* problem to use Boolean algebra efficiently.

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.

S*ound 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.