MATHEMATICAL THEORIES

William Bricken
March 1988

I recognize this is a quite foreign way of thinking, and that alone might disqualify it for serious consideration, but this form of model building will make possible what every good computer-aided-design (CAD) system will be expected to provide in the next decade, the ability to refer to constructions conceptually, rather than treating them as drawings without meaning (or at least as drawings that the user furnishes the meaning for rather than the software).

META-MODEL

Conceptual Structure ==> Mathematical Structure ==> Implementation Structure

where "==>" means dependent upon (in the marked direction) and abstracted from (in the other direction).

What's happening here is that a computational philosophy is being specified. Its mathematical RISC architecture. Computation by substitution and pattern-matching only. You need to optimize the hell out of a few things, and there is no other core machinery. Is it object-oriented? Yep. Pattern-matchers are all located as filters in the input and output handlers of each Class. Substitution is another word for message-passing. "Put yourself in my place." Is it a good idea? Take a close look at Mathematica.

An example

2D Geometry ==> Cartesian algebra ==> Arrays of coefficients

The Theory of Plane Geometry can be expressed in many different mathematical languages (like Cartesian algebra or Euclidean diagrammatics or matrices). What concerns me is that the *idea* of a Theory of Geometry is lacking in CAD. We do Matrix Theory well, and Dihedral Systems (rigid motions of plane figures), but we can't say a thing about similar triangles.

That is, CAD must be able to answer simple questions about a construction (Are these lines parallel? Is this angle the same as that angle? Will this peg fit inside that hole?) And the answers will have to come through symbolic reasoning rather than from direct measurement or from looking in the database.

MATHEMATICAL OBJECT-ORIENTED PROGRAMMING

Object Classes should reflect abstract organization as closely as possible, with Methods that formally transfer from the abstraction domain to the implementation domain. This layer of indirection can be compiled out later (for those concerned with efficiency), but *the layer of abstraction cannot be added later*. That is, if we don't build in the mathematical and conceptual abstraction now, we could loose everything later.

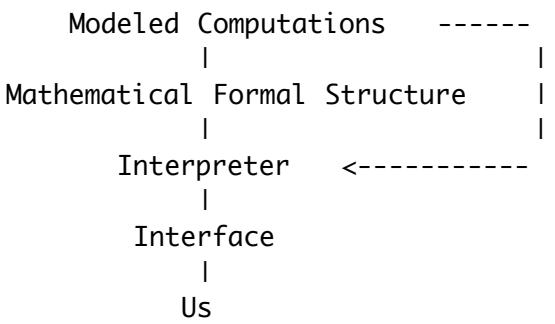
So here's the game: Whatever you think is the *conceptual domain* we are building software for, you must express it in one of the following mathematical languages. The conceptual domain is what you think the program should do (its specification so to speak), the intuitive notion behind the computational capability, the functionality of the software. This has nothing to do with implementation or machines or codes. It is what the user is provided by the software, hopefully with rigor.

Try specifying the mathematical organization of Plane Geometry. You may defer at any time to an implementation black-hole, otherwise known as hand-waving. The more waves, the less formal and the less rigorous (and the less valuable) your model is. Fortunately, the path from the mathematics to the implementation is known and relatively easy. It's specifying the map from concept to mathematics that is hard. This game is called Knowledge Engineering.

To translate between mathematics and object-classes:

-- An organization is a Class.
-- The language of an organization provides Type filters on local i/o buffers, or within the message manager.
-- The axioms of an organization are the Methods.
-- The transformation rules are the Computational Technique, the Disposition of a Class.
-- The functions are the message routing in Methods.
-- The relations are the Boolean tests in Methods.

THE PATTERN THAT CONNECTS



Yes, We are at the bottom, eagerly awaiting computational Results, separated from the Process only by an Interface.

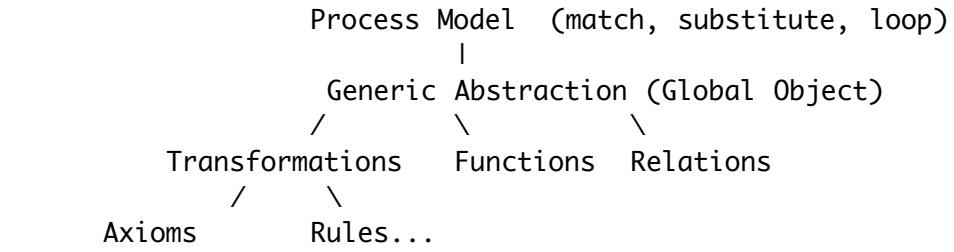
The Interface cleans the input, and kicks back to the user all unintelligible nonsense. This does not need to be crude, the Interface could gracefully help the user to put some computational sense into the input.

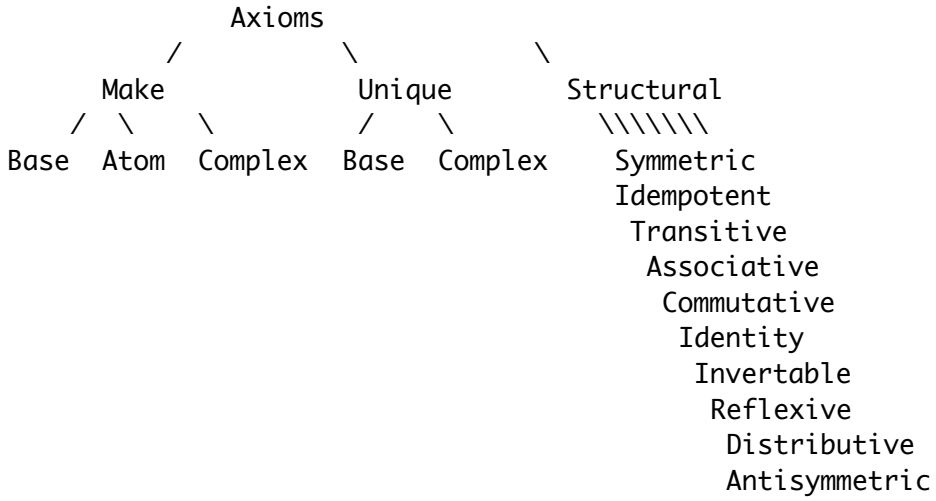
The Interpreter is part of a Loop (called Read-Eval-Print in LISP) which cycles clean input through the mathematical structure to the Implementation (the Modeled Computations). Basically, the Interpreter converts input into its mathematical form as expressed by the current implementation language.

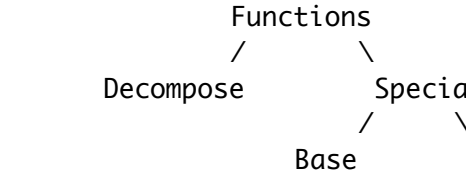
The Mathematical Structures activate Modeled Computations in lots of different ways, releasing results to the Interpreter, which collects and formats them to present to the Interface. Results need not be complete, the Interface can be capable of prettying up "I don't know" for the User. Sweet Interfaces will estimate time to completion of a task (by doing a structural analysis, omitting instances), if it's longer than you usually wait.

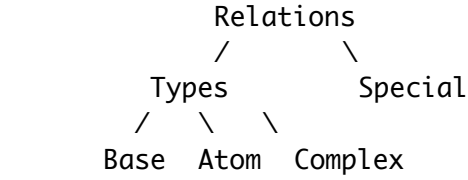
In the hierarchy that follows, the top is the Process Model, which corresponds to the Modeled Computations in the loop above. The Hierarchy itself embodies the concept of rigorous abstraction, describing the significant differences of almost all mathematical models. The leaves of the Hierarchy correspond to the complexity that the Interface must nicely present to the User.

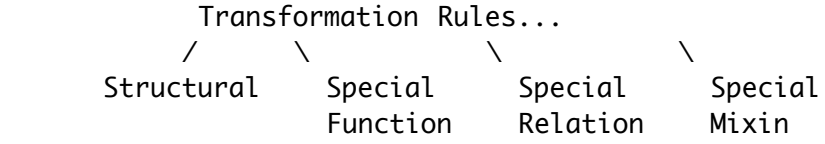
THE ABSTRACTION HIERARCHY











CIRCUS OF MATHEMATICAL ORGANIZATIONS

All of the following mathematical organizations have known efficient implementations. I've written in a mathematical notation. Stuff is from several books, particularly:

-- Manna and Waldinger, *The Logical Basis for Computer Programming*
-- Lucas, *Introduction to Abstract Mathematics*
-- Genesereth and Nilsson, *Logical Foundations of Artificial Intelligence*
-- Bavel, *Math Companion for Computer Science*
-- Abelson and Sussman, *Structure and Interpretation of Computer Programs*

The structure of mathematical organizations is presented for lots of theories (organizations). Missing are Modal and other special logics; Theories of Integers, Rationals, Reals, Irrationals, and Imaginarys; and most of the refined stuff.

Be Warned: I've used unconventional notation in some places, in an effort to remove a lot of confusion in current notation of abstract models.

Comments are separated from the components of structural theories by hash-marks.

Divergence into Biological Metaphors: The biological metaphor is that of *organization* and *structure*. The two concepts are distinct in living organisms and in abstract programs. Organization is what defines us as homo sapiens. Structure is what varies among us. Organization is the mathematical substance of a theory, the axioms that constitute the concept. Structure is what varies over languages and styles and syntaxes and personalities. Organization is the Class hierarchy; Structure is the run-time Instance configuration.

LOGIC

PROPOSITIONAL CALCULUS

Constants:
True, False

Objects:
Statements with no internal structure that may be True or False.

Variables:
p, q, r, ...

Functions:
none

Relations (connectives):

```
not p
p and q
p or q
p -> q            if p then q
p == q            p if-and-only-if q
if p then q else r
```

Expressions (sentences):
Statements in relation.

Interpretation:
An expression has the (global) value of the local value of the variables put in relation.

To interpret an expression: replace proposition names with truth values.

Properties:

Valid:	true for all interpretations.
Satisfiable:	true for some interpretation.
Contradictory:	true for no interpretation.
Logical implication:	if p is true then so is q.
Equivalent:	p and q have same truth value independent of interpretation.
Consistent:	all consequences are satisfiable.
Complete:	all valid consequences are reachable.

All properties are variations of Valid.

Computational Methods:

Truth tables:	Try all possible values.
Semantic trees:	Try all possible values of if branches.
Falsification:	Assume false value, show a contradiction.
Natural Deduction:	Apply inference rules.
Resolution:	Negate conclusion, show inconsistency by resolving.
Algebraic Deduction:	Apply transformations on algebraic expressions.
Boundary techniques:	Apply spatial reasoning to logic.

Properties of AND and OR:

Commutative:	(p and q) == (q and p)
Associative:	(p and (q and r)) == ((p and q) and r)
Transitive:	((p -> q) and (q -> r)) -> (p -> r)
Distributive:	p and (q or r) == (p or q) and (p or r)
Dual:	p or q == not ((not p) and (not q))

Utility:

program flow of control
simple models of type hierarchies
Egs: required classes to graduate
weaving patterns
decision structures,
arguments

PREDICATE LOGIC

Same as PROPOSITIONAL LOGIC, except the propositions can have internal structure, and quantification is added.

Constants:
  truth values: True, False
  simple objects: a, b, c, ...

Variables:
  x, y, ...

Functions:
  F[term1, term2, ...] arbitrary, determined by a specific domain

Relations (predicates):
  P[term1, term2, ...] arbitrary, from a domain.

Special relations (quantifiers):
  All: for all x, sentence A[x] is true -- assumed by algebraic context
  Some: for some x, sentence A[x] is true -- written as Ex

Composite objects (terms):
  constants, variables, functions in composition.

Expressions (sentences):
  Terms and their logical relationships

Axioms:
  Those of Propositional Logic and those from specific domains.
  Inference Systems, such as
    Exhaustive evaluation
    Natural deduction
    Resolution
    Equational substitution

STRUCTURAL THEORIES

PREDICATE LOGIC is enough for most purposes. Theories produce a subset of predicate logic by specifying specific objects, functions and relations, that is, a specific language.

Example: A THEORY OF 2D-POINTS

Constant:

o the Origin

Objects:

a, b ,c, ... POINTS, each composed of two REAL numbers, x and y.
a = (x,y)

Functions:

X[a] = x
Y[a] = y
Scale[a, n] = (nx,ny)
Transpose[a] = (y,x)
Distance[a, b] = b - a where "-" is a distance computing function

Relations:

Point[a]
Equal[a, b] = (a.x = b.x) and (a.y = b.y)

Axioms:

Generate base: Point[o]
Generate point: Point[a] = (x,y)

Unique base: o = (0,0)
Unique point: (a = b) == (a.x = b.x) and (a.y = b.y)

Decompose: Point[a] = (X[a],Y[a])
etc.

The Theory game requires everything to be explicit. If it's not in the language, it's not in the universe. If you want other things in your THEORY OF POINTS, add them explicitly.

Example: A THEORY OF IMMEDIATE FAMILY RELATIONS

A concrete example.

Objects:

Ann, Bob, Carl, ...	fathers and mothers

Variables:

x, y, ...

Functions:

Father[x]	father of x
Mother[x]	mother of x

Relations:

(x P y)	x IsTheParentOf y
(x GF y)	x IsTheGrandfatherOf y
(x GM y)	x IsTheGrandmotherOf y

Axioms:

(Father[x] P x)	My father is my parent.
(Mother[x] P x)	My mother is my parent.
(x P y) -> (Father[x] GF y)	My parent's father is my grandfather.
(x P y) -> (Mother[x] GM y)	My parent's mother is my grandmother.

Just to illustrate that this is abstract, I'll change the words and the meaning without changing the structure of the organization. You can think of the labels as describing the structure of a family tree.

Objects:

a, b, c, ...	atoms

Variables:

x, y, ...

Functions:

UpLeft[x]	ToTheUpperLeftOf x
UpRight[x]	ToTheUpperRightOf x

Relations:

(x A y)	x IsAbove y
(x FL y)	x IsFarLeftBranchOf y
(x FR y)	x IsFarRightBranchOf y

Axioms:

```
(UpLeft[x] A x)               My upperleft is above me.
(UpRight[x] A x)              My upperright is above me.
(x A y) -> (UpLeft[x] FL y)   My above's upperleft is my farleftbranch.
(x A y) -> (UpRight[x] FR y)  My above's upperright is my farrightbranch.
```

The following additional structure gives us the power to compare and sort.

Relation:

```
(x < y)                       x IsLessThan y
```

Axioms:

Transitive:	`(x < y) and (y < z)  ->  (x < z)`
Irreflexive:	`not (x < x)`
Asymmetry:	`not ((x < y) and (y < x))`
Inverse:	`(x > y) == (y < x)`

Rules are called Axioms and Theorems, depending on which we wish to nominate as more important. We tend to focus on regularities in the structure of an organization, and give them long names.

We can change the structure by adding an inverse relation. We get to look at < from the other side. This is not a change in organization because the new thing is defined solely in terms of the old thing.

Don't think that `LessThan` really means numerical or alphanumeric `LessThan`, it could also mean `TreeLessThan`, referring to a position in a hierarchy; `GradientLessThan`, referring to a gradient over a terrain; `NetLessThan`, referring to a path over an acyclic graph, etc.

`IsLessThan` is actually an arbitrary test. We put the meaning into < by attaching code that implements it. Might be `ASCIILessThan` for Alphanumeric symbols, `NumericalLessThan` for Integers or Reals, etc. The level of abstraction closest to the machine architecture is the place to stop compiling syntax. The level of abstraction closest to the cognitive model is the place to start defining syntax.

Of course, we can't use *any* test, the test must conform to some rules. The rules define the meaning of <, regardless of the interpretation we place on the symbol.

In reference to the idea of a strict partial ordering, the organization is the set of defining objects, functions, and relations (the expressions in the language) *and* the axioms that permit transformation of these expressions. The structure is the way we interpret and implement the organization. If we change the rules, we change the meaning of `LessThan`, we change its organization. But when organization is changed, so is the "its-ness", the changed thing is no

longer comparable to its previous self. We are free to change the structure of `LessThan`, by twiddling its implementation or how we interpret and use it for example, without worrying about changing its functional organization, its abstraction, its conceptual model, its reliability and rigor. Similarly, by debugging a Class hierarchy we accrue two major benefits:

Error Partition: Run-time errors are not due to the organization of the program; they are due to poorly described instance configurations.

Abstraction: We can compute over abstract patterns, solely with the Class hierarchy, without relying on instance binding. This provides very efficient type and structure checking and minimizes instance creation (ie: efficient structure sharing in memory). #

THEORY OF EQUIVALENCE RELATIONS

A generalized notion of good ol' Equal. Smart use of equality defines the state-of-the-art in implementation techniques.

Objects:

any domain

Relation:

(x = y)	x IsEqualTo y

Axioms:

Transitive:	(x = y) and (y = z) -> (x = z)
Symmetric:	(x = y) == (y = x)
Reflexive:	(x = x)

Rules:

Double Transitive:	(x = y) and (x = z) and (y = w) -> (z = w)

Computational Technique:

Structural substitution:	(x = y) -> (A[..., x, ...] = A[..., y, ...])

In particular,

Substitution in terms:	(x = y) -> (F[..., x, ...] = F[..., y, ...])
Substitution in sentences:	(x = y) -> (S[x] == S[y])
Structural replacement:	((x = y) -> S[x]) == S[y]

In general,

((x1 = y1) and (x2 = y2) and ...) -> S[x1,x2,...]) == S[y1,y2,...]

Existential structural replacement:	((Ex = y) and S[Ex]) == S[y]

When we add Equals into an existing theory, we add a lot of computational technique, and enter the familiar domains of arithmetic and algebra. But why limit this powerful computational technique to numbers? What we have here is *arbitrary algebras*. The A can be any function, any relation. That's why algebra is fun. We've already been using algebraic logic; Substitution and Replacement are described using logical equality, == instead of if-and-only-if.

THEORY OF STRICT PARTIAL ORDERINGS

Objects:
any non-categorical domain

Functions:
none

Relation:

(x < y)	x IsLessThan y

Axioms:

Transitive:	(x < y) and (y < z) -> (x < z)
Irreflexive:	not (x < x)
Asymmetry:	not ((x < y) and (y < x))
Inverse:	(x > y) == (y < x)

THEORY OF WEAK PARTIAL ORDERINGS

Relation:

(x ≤ y)	x IsLessThanOrEqualTo y

Axioms:

Transitive:	(x ≤ y) and (y ≤ z) -> (x ≤ z)
Reflexive:	(x ≤ x)
Antisymmetric:	(x ≤ y) and (y ≤ x) == (x = y)
Inverse:	(x ≥ y) == (y ≤ x)

THEORY OF ASSOCIATED RELATIONS

Relations:

=	IsEqual
<	IsLessThan
≤	IsLessThanOrEqualTo

Axioms:

Irreflexive restriction:	(x < y) == (x ≤ y) and not (x = y)
Reflexive closure:	(x ≤ y) == (x < y) or (x = y)

Rules:

Irreflexive:	not (x < x)
Reflexive:	(x ≤ x)
Totality:	(x < y) or (y < x) or (x = y)

Utility: sorting and comparing.

THEORY OF GROUPS

Axioms:

The three for =	
Transitive:	`(x = y) and (y = z)  ->  (x = z)`
Symmetric:	`(x = y) == (y = x)`
Reflexive:	`(x = x)`
Identity:	`(x G e) = e`
Inverse:	`(x G x') = e`
Associativity:	`((x G y) G z) = (x G (y G z))`

A group is an organization that has a special identity element that dominates other arbitrary elements, and a special inverse element for every arbitrary element, so that the element grouped with its inverse yields the identity.

Because we have equality, we can do substitution in general. #

Computational techniques:

Substitute:

```
(x = y) == ((x G z) = (y G z))
(x = y) == ((z G x) = (z G y))
(x = y)  ->  (x' = y')
```

Interpretations:

```
Let the Group operator be numerical +.
      e is 0; x' is -x.
Let the Group be numerical *.
      e is 1, x' is 1/x
```

Utility:

This theory includes Rotation, Translation, and Scaling of 3D bodies, regardless of whether the operations are implemented by matrices, quaternions, turtles, or Cartesian algebra.

Here is a clear case of levels of abstraction:

Rotation => Groups => Matrices => Lists => Arrays

Each organization to the right is an implementation of the one on the left. Abstractly:

Abstract Concept ==>
Abstract Theory ==>
Applied Theory ==>
Representation ==>
Implementation #

THEORY OF ORDERED PAIRS

Whooah! PAIRS can be interpreted as POINTS. Could all this really be useful? Whoah! PAIRS can be interpreted as LINES, as PAIRS of POINTS. Does this mean that we can get Geometry out of one system instead of several? Whoah! PAIRS can be interpreted as LISTS. Does this mean we get a software representation system too? Whoah! PAIRS can be interpreted as CONS-CELLS. We also get an implementation architecture? Yep, folks, that's what John McCarthy thought in the Fifties, when he invented LISP.

Objects:

```
      a,b,c,...                              PAIRS
            composed of       x,y,...        ATOMS
```

Functions:

```
  Constructor:     {x1,x2}                  to pair up
  First:           First[{x1,x2}] = x1
  Second:          Second[{x1,x2}] = x2
```

The constructor could be written in functional notation as Pair[x,y], or in infix notation as (x . y), or as above, in delimiting notation. A pair is two components within a parenthesis. Boundary notation is nice for patterns and constructors.

Relations:

```
      Atom[x]     x is on the AtomList or has the organization of an ATOM
      Pair[a]     a is on the PairList or has the organization of a PAIR
```

Note that if anything does not have the organization of an ATOM or a PAIR, then that something is outside the domain of the theory. There is only one sensible way to handle unknown structures, *treat them literally*, they are what they are written as. This means that a typing error should not necessarily stop execution. It could be passed as a literal into a context where it makes sense.

Operators should be overloaded in such a way that they work for any arbitrary organization, treating some with known transformations, others as non-transformable literals. #

Axioms:

```
  Generate pair:   Pair[a] == Atom[a.x1] and Atom[a.x2] and (a = {x1,x2})
  Disjoint:        not (Atom[x] and Pair[x])
  Unique:          {x1,x2} = {x3,x4}  ==  (x1 = x3) and (x2 = x4)
```

The Generate-pair Axiom spells out explicitly what the organization of a PAIR is, and is sufficient to assure a reliable implementation. Note the use of patterns in the definition, it can all be checked syntactically. Because patterns address a symbolic domain, the issue of binding regimes is irrelevant.

The Disjoint Axiom should be built directly into the labeling as:

```
{a,b,...} intersection {x,y,...} = { }
```

Non-intersecting name spaces are critical to any implementation.

The Uniqueness Axiom assures that pairs can be disassembled into components. This is most convenient, cause type checking and evaluation can be postponed until pattern criteria are fully satisfied. The responsibility for type checking should be local to the input and/or output handlers of a Class. In a PAIR {x1,x2}, the curly braces boundary is responsible for making sure that x1 and x2 are Atoms. That is, by writing {...}, we are also writing Atom[contents] = true. #

Structure:

```
Pair[a]  ->  Atom[First[a]] and Atom[Second[a]]
```

Computational Techniques:

Substitution: `(x1 = x3)  ->  {x1,x2} = {x3,x2}`

Decomposition: `Pair[a]  ->  a = {First[a], Second[a]}`

This lets the pattern matcher disassemble pairs.

THEORIES WITH LOOPS

We began with propositional logic, which provides programming control structure. Then we added a few theories that defined generic mathematical organizations for sorting and grouping. Now we add program iteration, also known as recursion, looping, and sequencing. (Iterate-one-time defines a sequence of instructions.)

At last! To satisfy the lust for numbers: #

THEORY OF NONNEGATIVE INTEGERS

Constant:
`0`
Constructor Function:
`x+`
Relation:
`Integer[x]`
Mixin equality:
`=`

Terms:
`0, 0+, 0++, 0+++. ...`

To shorten the notation, we count the number of `+`'s and call the term that number, to get `0, 1, 2, ...` (This is an example of swapping computation for definition.)

Axioms:

Generate base:	`Integer[0]`
Generate successor:	`Integer[x+]`
Unique zero:	`not (x+ = 0)`
Unique successor:	`(x+ = y+)  ->  (x = y)`

Computational Technique (Induction) (Loop):
`S[0] and (S[x] -> S[x+])  ->  S[x]`

Proof and Computation are defined by showing a base case, `S[0] == true`, and by assuming a general case, `S[x]`, and then showing the next case, `S[x+] == true`. This works because we know we can iterate over all numbers from `0` to `x` using only the stepping function `+`.

In recursive programming we do exactly the same thing (but usually in reverse, using a inverse theory that has Predecessor x- rather than Successor `x+`. For example, to sum, we exit at a base case, and iterate over each of the others explicitly:

```
Sum[x]  ==  if (x = 0) then 0 else x + Sum[x-]
```

The computational technique can also be recognized as iteration:

```
Sum[x]  ==  loop from 0 to x doing  (result := result + x)
```

Now I know that the machine does all this stuff reliably and it seems silly to spend so much time on it, *but* the conventional machine requires instances, it needs bound variables to work. The above techniques work on symbols, variables, unknowns, as well as on values, instances, knowns. Besides, it's not what the machine does that matters to us, it's what we understand about what the machine does from looking at i/o. If you're serious about low level objects, here's how integer objects manage to count. They don't flip bits, they pass "Add yourself to this" messages. #

Computational techniques:
  Substitution in general
  Decomposition: (x = 0) or (x = y+)

The Decomposition rule lets us define x- for all x except 0. It does this by introducing an arbitrary y.

Now we can expand to include addition and multiplication: #

Axioms for Addition:
  Addition base: (x + 0) = x
  Addition loop: (x + y+) = (x + y)+

Rules for Addition:
```
Integer[(x + y)]
(x + 1) = x+
```

Computational techniques:
  Substitution
```
(x + z) = (y + z)  ==  (x = y)
(x + y) = 0  == (x = 0) and (y = 0)
```

Axioms for Multiplication:
  Multiplication base: (x * 0) = 0
  Multiplication loop: (x * y+) = (x * y) + x

Rules for Multiplication:
```
(x * 1) = x
```

When x and y are different structures (the theory is applied to different domains), + and * have different definitions. The implementation changes (overloading) but the abstract organization stays the same.

I'm starting to abbreviate here, omitting the things that get inherited from other theories, and that are very much like theories previously described. Don't forget that there are an infinite number of rules, and substitution creates an infinite number of expressions, so let's just concentrate on the essentials. #

Mixin Associative Ordering:

```
<,  =,  ≤
```

Mix-in means build a world with both theories, explicitly coupling the theories with mixed rules. The world is not organizationally different because the theories maintain their unique organization while working in concert. This is how to achieve modularity of object-oriented systems. Coupling rules specify communication protocols.

Structural extensions:

```
Positive[x]  ==  Integer[x] and not (x = 0)
Minimum[x,y]  ==  (if (x <= y) then y else x)

Predecessor:          (x + 1)- = x
Subtraction base:     (x - 0) = x
Subtraction loop:     (x+ - y+) = (x - y)
                      (x + y) - y = x

Quotient base:        (x < y)  ->  (x/y = 0)
Quotient loop:        (x + y)/y = x/y + 1

Remainder base:       (x < y)  ->  (Remains[x,y] = x)
Remainder loop:       Remains[(x + y), y] = Remains[x,y]

Quotient-Remainder:   (x = (y*(x/y)) + Remains[x,y]) and (Remains[x,y] < y)

Divides:              (x D y) == (x * z = y) and Integer[z]
                      (x D y) == (Remains[y,x] = 0)
Multiply-divide:      (x D y) or (x D z)  ->  (x D y*z)

GreatestCommonDivisor Base:   Gcd[x,0] = x
GreatestCommonDivisor Loop:   (y = 0) or Gcd[x,y] = Gcd[y, Remains[x,y]]

    (Gcd[x,y] D x)  and  (Gcd[x,y] D y)
    (z D x) and (z D y)  ->  (z D Gcd[x,y])
```

In Divides, Integer takes care of the existence of the z, cause

```
Integer[non-existing integer] == false
```

This could be written as

```
(x D y) == (x * Ez = y)
```

Remember that the above extensions are to a theory of `NONNEGATIVE INTEGERS`, they take different forms for different domains, but of course we keep the syntax consistent by operator overloading.

OK so what about strings? #

THEORY OF STRINGS

By now the pattern that connects is emerging. Let's be succinct about this one.

Constant:

~	the empty string.

Characters:

u, v, ...

String variables:

x, y, ...

Generator Function:

.	to prefix

Relations:

Char[x]
String[x]

Axioms:

Generation base:	String[~]
Generation character:	Char[x] -> String[x]
Generation string:	String[(u . x)]
Unique base:	not (u . x) = ~
Unique concatenate:	(u . x) = (v . y) == (u = v) and (x = y)
Character equality:	(u . ~) = u
Loop:	S[~] and (S[x] -> S[(u . x)]) -> S[x]

Substitution in general

Functions:

Head[(u . x)] = u
Tail[(u . x)] = x

Rules:

Decomposition:	(x = ~) or x = (u . y)
Functional decomposition:	(x = ~) or x = (Head[x] . Tail[x])

Functions:

Concatenate base:	~ * x = x
Concatenate loop:	(u . x) * y = u . (x * y)
Reverse base:	Reverse[~] = ~
Reverse loop:	Reverse[(u . x)] = Reverse[x] * u

Rules:

```
Reverse[(x * y)] = Reverse[y] * Reverse[x]

End base:       (x IsAtTheEndOf ~)  ==  (x = ~)
End loop:       (y = ~) or
                      (x IsAtTheEndOf y == (x = y)) or
                            (x IsAtTheEndOf Tail[y])

Substring:      (x IsASubstringOf y) == (z1 * x * z2)
```

Mix-in Nonnegative Integers:

```
Length base:    Length[~] = 0
Length loop:    Length[(u . x)] = Length[x] + 1

Length-concatenate:   Length[(x * y)] = Length[x] + Length[y]

    not (Integer[x] and String[x])
```

THEORY OF TREES

Relations:

```
Atom[x]
Tree[x]
```

Constructor Function:

```
(x . y)                          to build
```

Axioms:

Generation atom:	`Atom[x]  -> Tree[x]`
Generation tree:	`Tree[(x . y)]`
Unique atom:	`not Atom[(x . y)]`
Unique build:	`((x1 . x2) = (y1 . y2))  ==  ((x1 = y1) and (x2 = y2))`

Rules:

Decomposition:	`Atom[x]  or  x = (x1 . x2)`

Functions:

```
Right[(x . y)] = y
Left[(x . y)] = x
```

Size base:	`Size[u] = 1`
Size loop:	`Size[(x . y)] = Size[x] + Size[y] + 1`
Tips base:	`Tips[u] = 1`
Tips loop:	`Tips[(x . y)] = Tips[x] + Tips[y]`
Depth base:	`Depth[u] = 1`
Depth loop:	`Depth[(x . y)] = Max[ Depth[x], Depth[y] ] + 1`

I've left out a lot of rules and stuff, the idea here is that TREES look a lot like STRINGS look a lot like LISTS:

Complex object:	`TREE`	`STRING`	`LIST`
Void:	none	~	`[]`
Base object:	Atom	Character	Atom
Constructor:	`.`	`.`	`.`
Constructor name:	build	concatenate	insert
Decomposer:	`(x . y)`	`(x . y)`	`[x . y]`
FrontAccess:	Right	Head	Head
BackAccess:	Left	Tail	Tail

But each has structures that make sense to that theory only. The Depth of a STRING is silly, as is the Length of a TREE.

There's not a lot of new meta-stuff coming, we've gotten to a relatively stable structure of abstract mathematical organizations. Notice how LISTS can be snuck in with only the surprises listed below. #

THEORY OF LISTS

TREES and NESTED LISTS are the same organization.

Relations:

Atlist[x] == Atom[x] or List[x]

Member base: not (u M [])
Member loop: (u M (v . x)) == (u = v) or (u M x)

Rule:

Atom[x] and List[x] -> (x = [])

Functions:

Append base: [] + x = x
Append loop: (u . x) + y = u . (x + y)

THEORY OF SETS

Constant:

```
{}, the empty set
```

Atom Variables:

```
u, v, ...
```

Set Variables:

```
x, y, ...
```

Relations:

```
Atom[u]
Set[x]
(u M x)                 u IsAMemberOf x
```

Constructor Function:

```
u{x}                    Insert u into x
```

Warning: I'm using the Set Theory curly brace as a boundary operator, which is not standard practice. Remember that it is *only* notation. We can write whatever we wish, and give it the meaning "Insert u into the Set x". So let's overload the boundary with the operation, cause then we only need to change our *reading* habits, rather than adding new notation. As is standard practice, what is written (the representation) is what is *before* computational action is taken. The u is outside the {x}, as in u{x}. The process of insertion then becomes u{x} => {u,x}. The boundary can enforce entry requirements like proper typing and u not already in x.

Technically, this change is one of redefining the representation of the empty set. A Constant is an Operation with no arguments. The representation {} now means to insert nothing into nothing, that is to do nothing at all. Once we start down this representational path, we see that the definition of Insert is a bit verbose. It suffices to say u{} => {u} #

Axioms:

```
  Generate base:   Set[{}]
  Generate set:    Set[u{x}]

  Member base:     not (u M { })
  Member loop:     (u M {v, x}) == (u = v) or (u M x)

  Unique atoms:    u{u{x}} = u{x}              # Multiciplicity is irrelevant #
  Exchange:        u{v{x}} = v{u{x}}           # Order is irrelevant #

  Associative
  Commutative
  Distributive
```

Wow! The same Set can be constructed in many ways, (loosely, a+b+c or b+a+c or ...) which means there will be no uniqueness axioms for set construction. This freedom from *linear organization* makes Set Theory the first theory that is truly multi-dimensional. Freedom to construct and access Set members non-linearily is analogous to using hash-tables instead of linear searching. This theory is critical for any parallel implementation. It is also expensive for a serial processor to implement correctly. That is, serial processors invariably map SETS onto more restricted theories, such as LISTS and ARRAYS. Any seriously optimized code will optimize the implementation of SETS.

Relations:

Component: (u M u{x})

```
Choice[x]
Rest[x] = x - Choice[x]
```

Talking about Choice is difficult in a linear language, ChooseOneOf[{a, b, c}] can return any one of them. We have to say it equals (a or b or c), which is more tedious than just writing out the Set. We almost always have a reason for choosing, and so Choice comes with a built in Filter. If you really don't care, then it will usually return the atom that is easiest to get (not a random atom, that's computationally expensive, and even if the user doesn't care, the implementer does). So:

```
Choice[ Filter[S], Set[x] ]  =  Choice[ SFilter[x] ]
```

#

Rules:

Unique base:	not (u{x} = {})
Absorption:	(u M x) -> (u{x} = x)
Equality:	(x = y) == ((u M x) == (u M y))
Decomposition base:	not (x = {}) == (Eu M x)
Decomposition member:	(w M x) -> (x = w{Ez}) and not (w M z)
Decomposition set:	(x = {}) or ((x = Ew{Ez}) and not (w M z))

Functions:

Decompose function:	(x = {}) or Choice[x]{ Rest[x] }
Nonmember:	(x = {}) or not (Choice[x] M Rest[x])
Sort:	(x = {}) or (Atom[Choice[x]] and Set[Rest[x]])
Union base:	({} U y) = y
Union loop:	(u{x} U y) = u{ (x U y) }
Union member:	u M (x U y) == (u M x) or (u M y)
Intersect base:	({} I y) = {}
Intersect loop:	(u{x} I y) = (if (u M y) then u{x I y} else (x I y))

Intersect member: `u M (x I y) == (u M x) and (u M y)`

`Disjoint[x,y] == ((x I y) = {})`

`U` and `I` are associative, commutative, distributive.

The discerning reader will have noticed that this theory seems to map onto `PROPOSITIONAL CALCULUS`, the first theory in the circus. Note how `Union-member` subtly converts `U` to logical `OR`. Note how `Intersect-member` converts `I` to logical `AND`. That's because they are exactly the same organization being applied to different types of Atoms. The same Class works for both Sets and Propositions.

Well, they're not exactly the same, since propositional rules are linear and set rules are not. But they are exactly the same, it's just that the people doing logic don't readily acknowledge (by their technical work) the folks doing algebra. #

Functions:

Delete base: `({} - u) = {}`
Delete loop: `(u{x} - v) = (if (u = v) then (x - v) else u{x - v})`

Difference base: `(x ~ {}) = {}`
Difference loop: `(x ~ u{y}) = (x - u) ~ y`

Relations:

Subset base: `({} ≤ y)`
Subset loop: `(u{x} ≤ y) == (u M y) and (x ≤ y)`

Subset is actually a mixin of the `WEAK PARTIAL ORDERING` theory. Proper subset is a mixin of the `STRONG PARTIAL ORDERING` theory.

Mixin Integer Theory:

Cardinality base: `Card[{}] = 0`
Cardinality loop: `(u M x) or Card[u{x}] = Card[x] + 1`

Cardinality union: `Card[(x U y)] + Card[(x I y)] = Card[x] + Card[y]`

`Singleton[x] == (x = Eu{})`

Cardinality singleton: `Singleton[x] == (Card[x] = 1)`

Set Constructors:

Base: `(Filter[u] In {}) = {}`
Loop:
```
(Filter[u] In v{x}) =
    (if Filter[v] then v{(Filter[u] In x)} else (Filter[u] In x))
```

Constructor Intersection: `(x I y) = Filter[u In x] In y`
Constructor Delete: `(x - v) = Filter[not (u = v)] In x`
Constructor Difference: `(x ~ y) = Filter[not (u In y)] In x`

The Set Constructor takes any expression in the language of sets, and builds a Set that is described by that expression. The function `Filter` takes a Set Closure (everything) and separates and collects the members that satisfy the specified expression (the Filter). This can be nicely implemented as a stream that furnishes values on command. Looks like this:

```
Stream[ Preprocess[ Filter[ Choice[x] ] ], Preprocess[ Rest[x] ] ]
```

The Stream consists of two elements, the first and second. The first is a single element selected from the Set `x`, and is ready to ship on demand. The second is the rest of Set `x`, ready to be selected from to replace the exiting first element. The Stream does not care when its empty, the error is in an object continuing to query a stream that has returned an empty reply. The stream might as well call the Resource Manager to come Garbage Collect it whenever it gets empty. That way, the dumb request for a confirmation of emptiness would fall into the Void.

Actually, the Set Constructor is the same as the Membership relation. Each implies the other. We can use the Set Constructor to define the rest of the theory, if we wish. #

THEORY OF BAGS

Just like the theory of SETS, but multiplicity counts, so the Unique-atom Axiom is repealed. Also known as MULTISETS.

New Axioms:
Unique: `if u{x} = u{y} then (x = y)`

Rules:
Count base: `(u C {}) = 0` # counts the occurrences of u in x #
Count positive loop: `(u C u{x}) = (u C x) + 1`
Count negative loop: `(u = v) or ((u C v{x}) = (u C x))`

Sum base: `({} + y) = y`
Sum loop: `(u{x} + y) = u{x + y}`

THE THEORY OF TUPLES

Some folks have been known to call tuples ARRAYS. Yuck, what a bizarre name.

Tuples are collections of elements, in order and with multiplicity. Tuples are a linear constriction of SETS, a simplification of LISTS in which nested lists are not allowed, and are similar to STRINGS, except that the single tuple <a> is different than the Atom a.

I mean what we have here is the same old axioms for

Generate	<a,b,...>
Unique	(a<> = b<>) == (<a> = <b>)
Member	(a M b)
Append	<a><b> = <a,b>

but here's a new one: #

Special Relations:

Same base:	Same[<>]	# All the same atom in the tuple. #
Same unit:	Same[u<>]	
Same loop:	Same[u<v<x>>] == (u = v) and Same[v<x>]	

Let's do array accessor functions. <x>n means the nth element of x. Like all good computer scientists, we start counting at 0.

Functions:

Element base:	u<x>0 = u
Element loop:	u<x>n+1 = <x>n

What a cop out. We sequentially run down the array to find an indexed element. What ever happened to direct access? What a good idea for an implementation of SETS!

Wanna get fancy with mixins? #

BAGTUPLES:	Remove ordering from tuples.
SETBAGS:	Remove duplicates from bags.
SETTUPLES:	Remove both ordering and duplicates from tuples.

Composition:

Settuple[x] = Setbag[Bagtuple[x]]

Relation:

Ordered base:	`Ordered[<>]`
Ordered single:	`Ordered[u<>]`
Ordered loop:	`Ordered[u<v<x>>] == (u =< v) and Ordered[v<x>]`

Function:

Sort base:	`Sort[<>]`
Sort loop:	`Sort[u<x>] = Insert[u, Sort[x]]`
Insert base:	`Insert[u, <>] = u<>`
Insert loop:	`Insert[u, v<x>] = (if (u =<= v) then u<v<x>> else v< Insert[u, x]`

`>)`

Rules for Sorting:

```
Ordered[ Sort[x] ]                  # That's both comforting and provable. #
Sort[ Sort[x] ] = Sort[x]           # We don't ever have to Sort twice. #
```

THEORY OF ENDINGS

Relation:

`Finis[circus-of-mathematical-organizations]`