COMMENTS ON CODING

William Bricken
June 1989

> *The code that implements a function is the documentation of the function's definition!*

MATHEMATICAL CLASSES

Here's what Mathematical Classes mean for an implementation architecture. Each class has a similar internal structure:

1. The Class handles nonexistence, by never creating a void instance. Could be implemented with a named-instance-table.

2. Base objects (named atoms) never initiate messages. They are the final resolution, and they do answer queries. When bound, they immediately Absorb into their functional context.

3. Complex objects are decomposed to base objects, by passing messages down a structural hierarchy.

4. Computation is done with equality comparisons, and with substitution. Equality of complex structures is done with pattern matching over structure. Substitution is initiated when pattern matching fails; it moves the structure towards a normal form. When a pattern match fails on a normal form, we have a result, and terminate execution.

5. Note that type-checking and syntax errors are filtered out by passing messages around the class hierarchy. Instances are consulted only to resolve questions, which is done by binding variables. Since instances exist only at run-time, programs can be debugged abstractly, at compile-time. Run-time errors indicate a world model error (an impossible drawing for the model), not a coding error.

TYPE CHECKING

Type checking is a limited form of CONSTRAINT REASONING. Why not permit arbitrary constraints as filters on i/o of an object. This is the Filter concept of Set Constructors.

RECURSIVE STYLE

We can use the Base and Loop definitions to form a recursive algorithm. The general pattern is:

```
Recur base:       F[x, base] = S[x]
Recur loop:       F[x, y-increment] = T[x, y, F[x, y]]
```

Here we identify a function with the base constant of a theory, and with one step of the constructor function (y-increment). We store the results of the base case as S[x], and the results of the loop case as T[x, y, recur]. Here's an example, the Factorial function:

```
Base:             Factorial[n, 1] = 1
Loop:             Factorial[n, (i + 1)] = n * Factorial[n, i]
```

So S[n] = 1 and T[n, i, recur] = (n * recur)

In PROLOG, we would submit the definitions in a slightly different syntax. In LISP we would write the two parts in a slightly different way:

```
(factorial n)  =def=  (fact n n)
(fact n i)     =def=  (if (= i 1) 1 (* n (fact n (- i 1))))
```

Having the index i to count things suggests an iterative form of Factorial:

```
(factorial n (setq result 1))  =def=
      (do  (from i = 1 to n) (setq result (* result i)))
```

With a generator stream and a set theory, we could be very efficient and free (but alas, many machine architectures don't like this way of doing it):

```
(factorial n)  =def=  (* (stream 1 to n))
```

The Stream's Filter is "Accept everything". The Choice function is free to get the easiest element at all times. Using a pattern matching syntax:

```
(factorial n)  =def=  {* 1..n}
```

The main reason to do it this way is to be ready for parallel processing. Then the stream generator can also generate in parallel. For example, Stream could Choice pairs of elements and give each pair to a different processor. This converts the algorithm from O(n) to O(log n). That is to say, if we model (and implement) *orderless concepts*, such as space, using linear models, such as LISTS, we fail to prepare for medium-range future equipment. If we implement orderless concepts as SETS, we can swap hardware architectures without changing the organization of our code. We degenerate SETS, for example, into LISTS, when thinking solely for serial processing.

PROGRAMMING STYLE

There is no substantive difference between declarative, functional, and object-oriented styles. They *are* very different because of very impure implementations that compromise the organization of each. For example, in a functional regime, arguments are passed by location. In an object regime, messages are sent by name. These two approaches are implemented differently (they are different structurally), but they embody the same organization (function invocation and composition).

The power of mathematical modeling is that just about anything that's possible to say is said concisely. We get an instructional sequence, and an explicit description, and the assurance of stepping between process and data with ease and dexterity. The benefits of using mathematical organization as a central abstraction partition between concept (whatever we mean by a user action) and implementation (whatever we mean by a machine action) are precisely those of portability, ease of maintenance, verifiability, power, preciseness, modelability, and all the other stuff that we dream of.

That is not to say that problems don't exist. (Take the previous double negative, for example.)

Problems, ordered by worseness:

1. This technique may seem totally unintelligible, a foreign language, a stark raving. It isn't, of course, (what it is is the formal basis of computation), but if you don't see algorithms and even pseudo-code in the Base and Loop definitions of everything, then this is not for you. The worst thing is that it's novel.

2. Of course folks have built these systems. Genesereth's MRS at Stanford is a prime example. They used to run slow but they don't run slow any more (MRS was written years ago). They are very explicit and they're marvelously well-adapted for multiple processes.

But this is supposed to be a problem, and it is: Its hard to Knowledge Engineer using only abstract organizations. You must be very savvy mathematically and in your model. Now Geometry2D and Geometry3D are excellent places to use abstract mathematics, and a lot has been known for over fifty years, but it will take effort to map all the techniques of CAD onto their foundations. There is a sort of conservation of effort: *everything is hard*. We've made implementation (and modularization and maintenance and debugging) a lot easier by placing restrictions on how we specify our conception of geometry. In return, we must be totally clear about all CAD models. We can no longer expect the user to furnish the meaning. (Of course, anyone can turn off modeling and draw in freeform modes.) Fortunately, we can also provide the tools for users to build their own models. An "inner core" of this architecture is the mathematical

Classes upon which we will have built CAD geometry. If a user needs to convert an environment from Euclidean to Lobachevskian, the "applications developer" can modify the Axiom of Parallel Lines from the 2D Axioms and plow on.

3. There are many models and tasks that would make this system sluggish and computation/message bound. There are many mis-implementations that would make this approach intolerably slow. This is true in any useful domain.

The real issue of computation speed vs implementation optimization is that we accept a tighter language in exchange for a understandable processing model. The architecture of the machine could be fitted to the architecture of the processing model to maximize computational efficiency. What *is* terrible is mismatched models and machines.