EQUIVALENCE

William Bricken
August 1994

Equivalence is a particular (therefore named) set of names, defined by

```
            a=a  is True

            a=b & b=c -> a=c

            a=b <-> b=a
```

This statement of the particularities of Equivalence is

1. set in a context of logic
2. assumed binary
3. assumed spatially ordered.

Classical representation has deep roots that are without utility.

Free the spatial context by building Symmetry into the representation

```
            =[ a b ]
```

means spatially disjoint relation. Symmetry is an access issue.

```
            =[ a b c ... ]
```

means cardinality independence. Transitivity is a collection issue.

```
            =[ a a ] ==  <some intention>
```

If information is generated in binary form, i.e. GET returns one thing at a time, then integrate binarity into the spatial representation. If to do that requires binary PUT, then so be it, but the rest is SET.

Identity is the most difficult to reduce, since logic is assumed under all three rules, as well as under the rule combinator, AND, and the rule evaluator, IsTrue.

Use

```
            ==
```

to mean Intention, the *MetaEquivalence*.

Within a boundary logical framework:

```
            =[A A]  ==  (  )
```

```
      Proof:
                  ( ((A) A) ((A) A) ) = (  )
```

which works whenever A includes names within a domain.

When A is Boolean, use ( ) as True and logical Infer as Pervasion, which says that Space is Disjunctive.

When A is Numerical, use ( ) as 1 and numerical LessThan as Pervasion, which is Space as Additive.

When A is Ring, use ( ) as Shift and cyclic Order as Pervasion, that is Space as Permutation.

Symmetry extends Pervasion into any access ordering:

```
            (  ( (=[A B]) =[B A] )  ( (=[B A]) =[A B] )  )   ==   (  )
```

Sure there is a preference, but that is independent of how we choose from the bag. Choice is the Identity axiom, why we have elected not to be able to distinguish two cardinally different things. Symmetry is built into Space as Independence of Access Criteria.

Transitivity, like Inference and Addition, is the workhorse of Equivalence.

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

is Equivalence-Pervasion (works at all depths, etc.)

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

is the logical interpretation/intention.

We use boundaries that build in Transitivity. The question is access, i.e. why have we found ourselves in the situation of having to move the results into new buckets? That is dynamic database transaction control.

What is the *intent* of saying

```
            =[A B]
```

Domains anchor intention. Each domain is a collection of grounds and things to do with them.

For the sake of convenience (parallelism, decomposability, bandwidth, abstraction), we want to have collections-of-grounds as a tool. We can then define all the things to do over collections, including the handy singleton-collection.

Things to do with collections-of-grounds:

get them	(accessors-of-collections-of-grounds)
have them meet some intention	(relations-between-collections-of-grounds)

Under the accessor model, Properties, Functions, Filters, Tests, Predicates and Attributes are all ways to get some ground. When and why you get it depends on the Intention.

Accessors have two states, opaque or transparent (quote or eval). fac[3] is opaque to the thing you get back. Function-evaluation makes it transparent, by going and getting the thing. The transparent form of fac[3] is 6.

The interplay between open and closed representations depends on processes available to GET. Both the structure of physical access to memory and the structure of the symbolic question asked of memory effect transaction times. When transaction times are variable, synchronization of queries gets important.

One tool to help manage disparate queries and orchestrated information gathering is Equivalence.

Establish Space to assert only Equivalence upon its members. (other egs: the Logical Disjunctive and the Numerical Additive) Provide it to all Domains, so it is an abstraction over names and accessors, it is a generally available relationship. The intention of the use of =[...] depends only on the other givens in the symbolic domain.

```
In summary,            =[...]  is defined by

                  a Space           =[  ]
                  an Intention      =[A A]
                  a Pervasion Rule (=[A B]) =[B C]  ==>  =[A B C]
```

Implementing Access as Pervasion:

```
            if  =[...]1  intersection =[...]2   IsNotEmpty,

                  then Set-combine:       (=[A B])=[B C]  ==>  =[A B C]

      (...)...
```

means "if any of me is in you, then we combine."

This operation is Unique-Name-Preservation, the only reason why Equivalence and representation get hard.

Finally, the Intent of Equivalence depends on the use of Domain names. Assume that names themselves are used in a generally similar manner across all domains of immediate interest. We then develop

```
      Equivalence-of-grounds:            =[a a]
```

What is the intention of carrying two symbols for the same thing? It is usually an excellent idea to use Unique Names (like numbers, days-of-week, propositions).

```
      Unique-Name-Preservation:          =[a a]  ==>  =[a]
```

Now consider base case, =[ ]

A convenient intention is to consider the Void to be equivalent to itself, placing us in a logical framework.

```
            =[ ]  ==  (  )
```

Singleton case,   =[a]

Here then is the *core of intent*. When we get to having a single ground, we can go no further. There are no more questions to ask:

```
            =[a]  ==  {halt, exit, i/o call}
```

When the goal is Named, that is: =[a] means a-is-the-goal, it can be externalized by turning the Equivalence-Relation into an Equivalence-Predicate (filter, test, query, etc)

```
            =[a B]  ==  =a[B]
```

This shift of usage is an intentional act, since "a" is chosen as the canonical name of the boundary. Eg:

```
            =[ fac[3] 6 ]
```

asks for information. But "6" is a ground, so all we are in doubt about is the fac[3] expression. So, intentionally

```
            =[ fac[3] 6 ]  ==  =6[ fac[3] ]
```

That is an open question, and we open it going to the definition of fac. Turns out it is sufficient to consider Definition as Intention. Every == sign can be read as "is-defined-as"

```
      fac[#]  ==

            if =1[#] then =1[ fac[#] ] else =[ fac[#] *[# fac[-1[#]]] ]

              ==

            ( ((=1[#]) =1[ fac[#] ]) (=1[#] =[ fac[#] *[# fac[-1[#]]] ]) )
```

This version of fac is very linear/recursive. With spatial *, the logical framework and control structure is exposed as irrelevant.

```
      fac[#]  == *[2 .. #]
```

To restate:

The singleton =[a] means RETURN the single ground, cause

```
            =[a]  ==  =a[ ]  ==  a
```

Note:

```
            =[a a]  ==  =a[a]  ==  (  )

            =[a B]  ==  =a[B]
```

Opaque structures can be arbitrary patterns, eg partially opaque as in

```
            *[# fac[-1[#]]]
```

Label the non-grounds with capitals A, B, C...

Logical link:

```
            =[a b]  ==  void
```

Which is to say, names and name equivalences cannot be asserted freely into Void-space.

So the final issue is

```
            =[A B]  and =[A B ...]:
```

This is the general case of the Equation.

```
            =[ A  ((A)) ]  ==  (  )
```