

Let S be the basic symbol set where S is an ordered set of symbols which correspond one-to-one with the effective atomic operational semantics of any programming language (of sufficient power and complexity, see [1].) That is, S represents the complete enumeration of the operational semantics of a programming language P . For example, S_1 would be the alphabetically first legitimate symbol in P , and S_i would be the symbol in P . A function is any compilable, syntactically correct, ordered collection of symbols in S . Let L be the language of all such functions, beginning with simple variations on the basic symbol set, whose rules of expansion we shall return to. Let L_i be the function in L , and the list of this set L is the complete enumeration of the basic names of all functions generated so far over S .

Let F be the ordered index of all functions in L which either (a) generate syntactically valid output from valid input, or (b) change the (intra)system state in any way. A structured partition of F is a collection of functions taken as methods of a class and which generate syntactically valid input and output. Let a component be any collection of functions which, taken as methods of a class, generate syntactically valid input and output and utilizes other member (or family) functions or variables. A component is by definition syntactically correct and has a rough, “machinistic” (part-to-whole) meaning.

We define a closed ontology O to be an ordered collection of micro-ontologies. A micro-ontology is a special kind of class structure in which an arbitrary assemblage of component names are related to other component-names/ around five categories:

(a) Individuals. An individual is just a named component; let I represent the group of all individuals belonging to the micro-ontology; the first individual is the name of the micro-ontology itself;

(b) Classes. A class C is an ordered group of components with some commonality. This commonality is represented as the name of the class. At the same time, the contents of C are entirely dictated the name of C (which is the addition of a new basic symbol in O):The name of C names the grouping-function of which C is the result;

(c) Attributes

(d) Relations

(e) Events: Events are interruptions of a process due to an input stream (words or images) which provoke some response; this response is re-encoded as a necessary relation between an “individual” of the class of “events”;

Let K be an open ontology, the set of all valid programs—just ordered collections of closed ontologies—correlated to themselves within a macro-ontology. We shall say K is equivalent to a simple agent. Let us define three rules which expand K (i.e., generate new functionalities.)...

(i) other:

a. the function calling this function is taken as the foundation for a new program;

b. four variations and cross-variation on this interaction

i. repetition/erasure

ii. extension/contraction

iii. displacement/reorganization

iv. integration/separation

(ii) same

a. this operation is repeated, treating the newly creating network of related programs as the other who calls

- b. (repeat the four variations)
- (iii) synthesis (new program generation)

[1] – The basic ‘trick’ here is that the recursive step appears in the definition itself, i.e., in the relation of the symbol set to itself. We’ll get into all this later, but the programming language in question has to at the least allow for recursion, the definition of an abstract class, and the treatment of program code as an atomic data type. Provided this, we have the raw mechanics we need to simulate a heterogeneous network of communicative, self-programming agents—which allows for the multiple social instantiations of linguistically competent agency. Treating the program code as a code allows the computer to treat any code as a code—i.e., gradually and smoothly increase its facility with a symbol set by using it, experimenting with it, etc.

(c) Fractal Ontology, 2007