Electronic Delivery Cover Sheet

Walter Fontana
Theoretical Division, MS-B213; Center for Nonlinear Studies, MS-B258, Los Alamos National Laboratory, Los Alamos, New Mexico 87545 USA; and Santa Fe Institute, 1120 Canyon Road, Santa Fe, New Mexico 87501 USA

Algorithmic Chemistry

In this paper complex adaptive systems are defined by a loop in which objects encode functions that act on these objects. A model for this loop is presented. It uses a simple recursive formal language, derived from the λ-calculus, to provide a semantics that maps character strings into functions that manipulate symbols on strings. The interaction between two functions, or algorithms, is defined naturally within the language through function composition, and results in the production of a new function. An iterated map acting on sets of functions and a corresponding graph representation are defined. Their properties are useful to discuss the behavior of a fixed-size ensemble of randomly interacting functions. This "function gas," or "Turing gas," is studied under various conditions, and evolves cooperative interaction patterns of considerable intricacy. These patterns adapt under the influence of perturbations consisting in the addition of new random functions to the system. Different organizations emerge depending on the availability of self-replicators.

Artificial Life II, SFI Studies in the Sciences of Complexity, vol. X, edited by C. G. Langton, C. Taylor, J. D. Farmer, & S. Rasmussen, Addison-Wesley, 1991

While the logical structure of Darwinism seems secure, this should not be taken to imply that Darwinism, when expanded to encompass hierarchical considerations, will not be found to possess a mathematical structure previously unsuspected.

—Leo W. Buss[3]
"The Evolution of Individuality"

1. WHAT THIS IS ALL ABOUT

1.1 INNOVATION

The evolution of living systems involves, by definition, the notion of innovation. The appearance of novelty occurs at many scales ranging from societies, to individuals, to cells, to genes, to molecules. Objects at all these scales typically exhibit a high diversity of interactions. Molecules, for example, encode a variety of interaction properties ranging from kinetic parameters to qualitative relations as expressed in chemical reactions. Most importantly, on all levels the interactions are constructive, in the sense that they enable, directly or indirectly, the formation of new objects.

Introducing a new object into such systems is, therefore, equivalent to the introduction of a variety of new relations. The newly created object interacts with other objects that are present, and spawns further interactions involving its products. This can lead to dramatic changes in the organization of a system. Physics usually does not consider situations of this kind.

Mechanisms for the generation of new objects from available ones can be placed between two extremes. New objects may be constructed by virtue of intrinsic and specific properties of the interacting objects, or they may arise purely by noise.

Chemistry is a clear-cut example for the former situation. The formation of a new molecular product in a chemical reaction is, to a large extent, a deterministic function of the interacting molecules (and thermodynamic variables as temperature, pressure, magnetization, stress, volume, etc.). The chemical properties of the reacting molecules give rise to a specific product.

In contrast, the generation of a new object through noise is well represented by the phenomenon of mutation. The "intended" reaction is the faithful replication of a DNA or RNA string, but a chance event, like the absorption of a high-frequency photon, causes a copying error. The resulting string may represent a new object. The point, however, is that the causing event stands in no further relation to its effect. Clearly, in any real situation even the non-random formation of products will be subject to noise.

This contribution considers the (idealized) non-random case: systems in which interactions among objects generate specific other objects. This case deserves particular interest because it isolates a situation in which objects encode operations whose targets are the same objects. As a result, self-organization occurs in both the space of objects and the space of functions associated with these. In a suitable representation, in which the construction of new objects is practically unbounded

(for example in chemistry), open-endedness might not only occur at the level of objects, but also—and most importantly—it can occur at the level of the relationships that arise among the objects. To put it with Max Delbrück[8]: "A mature physicist, acquainting himself for the first time with the problems of biology, is puzzled by the circumstance that there are no 'absolute phenomena' in biology. Everything is time-bound and space-bound." (Quotation taken from Mayr.[23])

1.2 FUNCTION

Physical objects that qualify as carriers of constructive interactions are far too complicated to model at present. Understanding from first principles how function is encoded into molecular objects is a major open problem, let alone how function is supported by supramolecular structures such as cells or cell-aggregates. In cases like societies and economies, it is even a major task to sensibly identify the various functional units.

How then can an innovative system based on non-random constructive interactions be modeled? How should an artificial world be defined that expresses in a transparent, tractable, and sufficient way

1. a combinatorial variety of structures, and
2. a mechanism by which these structures can manipulate each other?

A structure that manipulates another structure and outputs (uniquely) a further structure is a mathematical function. "Function" is a concept that is irreducible. It can be viewed in two ways:

1. A function as an applicative rule refers to the process—coded by a definition—of going from argument to value.
2. A function as a graph, refers to a set of ordered pairs such that, if $(x, y) \in f$ and if $(x, z) \in f$, then $y = z$.

The first view privileges the computational aspect of function. Mathematics provides, since 1936, through the works of Church,[5] Gödel,[13] and Turing,[33] a formalization of the intuitive notion of "effective procedure" in terms of a theory of a particular class of functions on the natural numbers: the partial recursive functions.

A formal system (like a computational language) secures the combinatorial variety of structures by a recursive definition of syntactically legal objects, and provides, through a few axioms, a semantics that defines the function, i.e., the manipulative part, associated with each object.

Church's λ-calculus[2,6] embodies in the most transparent way the basic ingredients for a simple and abstract model of a complex innovative system:

1. Functions are defined recursively in terms of other functions. This hierarchical construction is reflected on the syntactic level by defining legal objects as trees. This makes explicit that functions are "modular" objects, whose building blocks are again functions that can be freely recombined.

2. Objects in λ can serve both as arguments or as functions to be applied to these arguments.
3. There is no reference to any "machine" architecture.

It is remarkable that in the physical universe, so far, only the level of molecules has been observed to spontaneously support complex phenomena—as life—that are different, in kind, from the phenomena at all other levels. The intriguing feature at this level is the appearance of chemistry, eventually due to the distinguishing properties of the Coulomb force. The molecular level seems to be the first level in physics where a combinatorial variety of structures can "manipulate" each other in a way that is strikingly similar to "symbolic manipulation." It is at this level that the notion of "function" begins to emerge, and again in a strikingly similar sense as the intensional interpretation of function in mathematics: a term or an expression is given, and a function—a relation—can be associated with that expression beyond its literal meaning. Chemical systems, in sharp contrast to nuclear, atomic, or astronomic systems, generate a level at which a description in terms of functional interactions becomes more adequate than in terms of the fundamental forces involved.

1.3 ARTIFICIAL WORLDS BEYOND LOTKA-VOLTERRA

This contribution is intended to show that a model universe made up of "particles" that are functions in the sense of a (universal) formal language accomodates innovation in a very simple and straightforward way. Function composition induces a dynamics in the space of functions (see section 4). In addition, the interacting objects exhibit a time evolution in relative concentrations as a result of "mass" action kinetics. The interaction between these two dynamical systems—one concerning the evolution of a nonlinear dynamical system on a support, the other governing the change of that support—has been articulated in different contexts by Doyne Farmer,[11] Stuart Kauffman,[17] Richard Bagley,[1] Norman Packard,[27] Steen Rasmussen,[29] John Holland,[16] and probably directly or indirectly by many others as well.[24]

This section puts the nonlinear system into relation with previously studied equations of the Lotka-Volterra type. Suppose that we have an infinite population. This assumption makes the support-dynamics obsolete, but isolates the structure of the nonlinear system. Suppose further, that an object j interacts with $\alpha = 1, 2, \ldots$ and other objects $k, l, m, \ldots$ to produce an object i. Let $x_i(t) \in \mathbb{R}_0^+$ denote the frequency of i in the system at time t. Furthermore, let a_{ijk} be a coefficient that quantifies the change in i upon interaction of j with k. For example, a_{ijk} could be the probability by which i is produced given that an interaction occurs between j and k. In general, in $a_{ijklm\ldots z}$ the first index indicates the product object, and subsequent indices refer to the interacting objects. Then,

$$\dot{x}_i = \sum_{j,k} a_{ijk} x_j x_k + \sum_{j,k,l} a_{ijkl} x_j x_k x_l + \cdots - \Omega(t) x_i, \quad i = 1, 2, \ldots \tag{1}$$

The proportional dilution flow $\Omega(t)$ is chosen such that $\sum_i x_i(t) = 1$, in which case,

$$\Phi(t) = \sum_{i,j,k} a_{ijk} x_j x_k + \sum_{i,j,k,l} a_{ijkl} x_j x_k x_l + \cdots, \tag{2}$$

and the system is confined to a simplex. To remain as simple as possible objects can be restricted to binary interactions. Equation (1) then reduces to

$$\dot{x}_i = \sum_{j,k} a_{ijk} x_j x_k - x_i \sum_{r,s,t} a_{rst} x_s x_t, \quad i = 1, 2, \ldots. \tag{3}$$

An *ansatz* like Eq. (1) assumes mass action kinetics, and raises the question about the construction of objects that require the "simultaneous" interaction of many other objects (say, more than three). Can such a construction be broken up into a series of binary interactions? In other words, it is not obvious how third- and higher-order terms in Eq. (1) should be interpreted physically.

In the present work the objects are functions. A function particle $j(v_1, v_2, \ldots, v_\alpha)$ of α variables, then, interacts through function composition with α functions $k, l, m, \ldots$ (all of which with an arbitrary finite number of variables) to produce a function particle i (of some number β of variables), $i(v_1, v_2, \ldots, v_\beta) = j(k, l, m, \ldots)$. For the coefficients we have, $a_{ijk} = 1$ if j is a function of one variable and its action upon k (arbitrary finite number of variables) produces i, $i = j(k)$, and $a_{ijk} = 0$, otherwise. a_{ijkl} denotes the analogous production coefficient with j being a function of two variables acting upon k and l (in this order) to generate function i. In general, in $a_{ijklm\ldots z}$ the first index indicates the product object, the second index denotes the acting function, and subsequent indices refer to as many argument functions (of arbitrary finite number of variables) as the domain of definition of the acting function requires. Notice that since the objects are functions, uniqueness translates into

$$\sum_i a_{ijk} = 1, \sum_i a_{ijkl} = 1, \ldots, \tag{4}$$

i.e., a function j acting on argument k evaluates to a unique product i. It follows that $\Phi(t) = 1$, and considering only functions in one variable Eq. (3) simplifies further to[25]

$$\dot{x}_i = \sum_{j,k} a_{ijk} x_j x_k - x_i, \quad i = 1, 2, \ldots. \tag{5}$$

Systems consisting of objects that are copied by themselves (self-replicators), and/or by others arise in a variety of contexts, e.g., sociobiological game dynamics, ecology, economics, population genetics, and molecular evolution. The equation that represents the common thread in all those areas is the "replicator equation" (for an extensive survey see Hofbauer[15])

$$\dot{x}_i = x_i \left(\sum_j a_{ij} x_j - \sum_{r,s} a_{rs} x_r x_s \right), \quad i = 1, \ldots, n. \tag{6}$$

It has been shown[15] that a diffeomorphism converts the replicator equation in n variables into the Lotka-Volterra equation

$$\dot{y}_i = y_i(r_i + \sum_j a'_{ij} y_j) \quad i = 1, \ldots, n-1, \tag{7}$$

in $n-1$ variables on the positive orthant. Lotka-Volterra equations are widely used to model ecosystems.

Now observe that the replicator (or Lotka-Volterra) equation is a special case of Eq. (3), in which $k = i$, and all coefficients a_{rst} are of the form $a_{rsr} \equiv a_{rs}$. Since all i are replicated under the action of some j, $i = j(i)$, the first sum in Eq. (3) runs only over j, and, thus, becomes Eq. (6). Equation (3) is a generalization of the replicator equation, in that it drops the assumption that individual objects must be replicated.

Recently, Peter Stadler and Peter Schuster[32] have studied the replicator Eq. (6) including mutations. This means that object j copies object k, but makes an error with some probability distribution q_{ik}, thereby producing object i. The resulting equation is very similar to Eq. (3), but the coefficients a_{ijk} factorize into $a_{jk}q_{ik}$. The first factor describes the efficiency of the copy action of j upon k, the second factor describes the product. Notice that the interaction product does not depend on j, but only on i. It cannot depend on j, because the underlying assumption is that of a chance event representing an error in the reproduction of k: polymerases are not supposed to produce specific errors. This is not the case in Eq. (3), where the product i depends on both j and k. Stadler's and Schuster's system is one important example for the limiting case of a purely noise-induced production of new objects. The corollary is that, as noise tends to vanish the replicator Eq. (6) is regained, and, therefore, provides for a reference state that allows an analytical perturbation approach. The limiting case of a non-random production of new objects, as depicted by Eq. (3), has no reference state. This brings the above quotation from Delbrück back to one's mind.

Innovation requires a combinatorial variety of structures. In the case of computable functions over $\mathbb{N}$ this variety is countably infinite. In all finite-universe size limitations have to be imposed. But even very strong limitations cannot prevent the numbers involved to quickly exceed the material and temporal resources of many universes. The number of baryons in the universe is already matched by the number of variations of strings of length 80 over an alphabet of size 10. This calls for a stochastic description. The Turing gas described in section 5 is precisely a stochastic simulation of Eq. (5). Typical questions that arise are: Which sets of interaction patterns can coexist for how long under which conditions? How do once established interactions constrain the subsequent evolution of the system? How do mutually stabilizing interactions respond to perturbations consisting in the introduction of new interaction carriers? Does the motion of a finite system in interaction space exhibit attractors of the type featured by usual dynamical systems? How can cooperativity be characterized and classified?

The artificial worlds of Eq. (1) are specified by the coefficients $a_{ijk}, a_{ijkl}, \ldots$. These coefficients involve a concrete description of concepts like "object" and "interaction." The problem is to achieve a finite description of an infinity of possible structures that these objects might acquire. For most cases in chemistry, biology, or economy this is tantamount to a theory, or at least a model, of the appropriate entities.

The avenue taken here is to identify "object" with "function" and to use a theory of function, as Church's λ-calculus, to decide, i.e., to compute a_{ijk}, or equivalently to compute i, given j and k. This algorithmic toy-chemistry captures a constructively innovative system in a transparent way, but how does it connect to physics or biology? Stated differently, the question is: to what extent are pure functions the right objects to consider in biological systems?

A model-like the one proposed here, cannot provide much detailed information about a particular real complex system whose dynamics will highly depend on the physical realization of the objects as well as on the scheme by which the functions or interactions are encoded into these objects. The hope is that an abstraction cast purely in terms of functions, defines a level of description that enables a logical and mathematical characterization of patterns of physical organization. The conjecture is that a world of functions is indeed homomorphic to the real world. But still: how much—in the case of biological systems—can we abstract from the "hardware" until a theory loses any explanatory power? We don't know yet. This is tightly connected to deep problems of inference concerning artificial constructive worlds in general, and that go beyond the conventional problems in mathematical modeling. These issues are addressed by David Lane and John Holland within the economics and the adaptive computation programs at the Santa Fe Institute.[19]

1.4 ORGANIZATION OF THE PAPER

Section 2 summarizes the model; section 3 briefly describes the language used to provide the mapping from syntactically legal character strings to algorithms, i.e., functions, that operate on character strings. Section 4 introduces an iterated map acting on a set of functions and its graph representation. Some concepts are defined that are used in the discussion of computer experiments concerned with the behavior of an ensemble of functions that act upon each other under particular conditions: a "Turing gas," defined in section 5. Some results are presented and discussed in section 6. Section 7 summarizes, and section 8 concludes the paper with an outlook. A formal and detailed description of the computational language used to encode the functions is relegated to the appendices.

This contribution is a modified version of a paper submitted for publication to *Physica D*.[12]

2. THE MODEL

To set up a model that also provides a workbench for experimentation, a representation of functions along the lines of the λ-calculus is needed. The representation used here is a somewhat modified and extremely stripped-down version of a toy-model of pure LISP as defined in Chaitin.[4] In pure LISP a couple of functions are pre-defined (six in the present case). They represent primitive operations on trees (expressions), for example, joining trees or deleting subtrees. This speeds up and simplifies matters as compared to the λ-calculus, in which one starts from "absolute zero" using only application and substitution. For the sake of simplicity, only functions in one variable are considered. The language is briefly explained in section 3.

The model is built as follows:

1. Universe. A universe is defined through the λ-like language. The language specifies rules for building syntactically legal ("well-formed") objects and rules for interpreting these structures as functions. In this sense the language represents the "physics." Let the set of all objects be denoted by $\mathcal{F}$.
2. Interaction. Interaction among two objects, $f(x)$ and $g(x)$ is naturally induced by the language through function composition, $f(g(x))$. The evaluation of $f(g(x))$ results in a (possibly) new object $h(x)$. Interaction is clearly asymmetric. This can easily be repaired by symmetrizing. However, many objects like biological species or cell types (neurons, for example) interact in an asymmetric fashion. I chose to keep asymmetry.
   Note that "interaction" is just the name of a binary function $\phi(s,t)$ that sends any ordered pair of objects f and g into an object $h = \phi(f,g)$ representing the value of $f(g)$. More generally, $\phi(s,t) : \mathcal{F} \times \mathcal{F} \mapsto \mathcal{F}$ could be *any* computable function, not necessarily composition, although composition is the most natural choice. The point is that whatever the "interaction" function is chosen to be, it is itself evaluated according to the semantics of the language. Stated in terms of chemistry, it is the same chemistry that determines the properties of individual molecules and at the same time determines how two molecules interact.
3. Collision rule. While "interaction" is intrinsic to the universe as defined above, the collision rule is not. The collision rule specifies essentially three arbitrary aspects:

   a. What happens with f and g once they have interacted. These objects could be "used up," or they could be kept (information is not destroyed by its usage).

   b. What happens with the interaction product h. Some interactions produce objects that are bound to be inactive no matter with whom they collide. The so-called NIL function is such an object: it consists of an empty expression. Several other constructs have the same effect, like function expressions

that happen to lack any occurrence of the variable. In general, such products are ignored, and the collision among f and g is then termed "elastic"; otherwise, it is termed "reactive."

c. Computational limits. Function evaluation need not halt. The computation of a value could lead to infinite recursions. To avoid this, recursion limits, as well as memory and real-time limitations, have to be imposed. A collision has to terminate within some pre-specified limits; otherwise, the "value" consists in whatever has been computed until the limits have been hit.

The collision rule is very useful for introducing boundary conditions. For example, every collision resulting in the copy of one of the collision partners might be ignored. The definition of the language is not changed at all, but identity functions would have now been prevented from appearing in the universe.
In the following, it will be implied that the interaction among two objects has been "filtered" by the collision rule. That is, the collision of f and g is represented by $\Phi(f,g)$ that returns $h = \phi(f,g)$ if the collision rule accepts h (see item (b) above); otherwise, the pair (f,g) is not in the domain of Φ.

4. System. To investigate what happens once an ensemble of interacting function "particles" is generated, a "system" has to be defined. The remaining sections will briefly consider two systems:

a. An iterated map acting on sets of functions. Let $\mathcal{P}$ be the power set, $2^{\mathcal{F}}$, of the set of all functions $\mathcal{F}$. Note that $\mathcal{F}$ is countable infinite, but $\mathcal{P}$ is uncountable. Let $\mathcal{A}_i$ denote subsets of $\mathcal{F}$, and let $\Phi[\mathcal{A}]$ denote the set of functions obtained by all $|\mathcal{A}|^2$ pair interactions (i.e., pair collisions) $\Phi(i,k)$ in $\mathcal{A}$, $\Phi[\mathcal{A}] = \{j : j = \Phi(i,k), (i,k) \in \mathcal{A} \times \mathcal{A}\}$. The map M is defined as

$$M : \mathcal{P} \mapsto \mathcal{P},\ \mathcal{A}_{i+1} = \Phi[\mathcal{A}_i]. \tag{8}$$

Function composition induces a dynamics in the space of functions. This dynamics is captured by the above map M. An equivalent representation in terms of an interaction graph will be given in section 4.

b. A Turing gas. The Turing gas is a stochastic process that induces an additional dynamics over the nodes of an interaction graph. Stated informally, individual objects now acquire "concentrations" much like molecules in a test-tube mixture. However, the graph on which this process lives changes as reactive collisions occur. Section 6 will give a brief survey on experiments with the Turing gas.

3. THE LANGUAGE

The language used to express the algorithms is closely related to, so-called, pure LISP, and in particular to a toy version designed by Gregory Chaitin.[4] Using a λ-calculus type of language turns out to be critical: it is a functional, as opposed to a procedural, programming language.

The procedural programming mode is the standard approach in traditional von-Neumann languages like Pascal, FORTRAN, or C. In this mode a program is a step-by-step specification of an algorithm. Instructions are executed in order, and the state of a program at any point of its execution is determined by the values of various variables in use. Procedural programming uses elements like conditionals and iterations to control the execution flow. At any point during execution, a program will have many local variables and various control structures in use. It is precisely this property that constrains a procedural program syntactically and semantically, making it very difficult to plug in random pieces of code.

The basic idea of functional programming[22] is to specify an algorithm by nesting functions. A function manipulates a string of characters. At any time only one particular function is active, and the string that this function manipulates is called the "current expression." When the currently active function has terminated its operations, the current expression is passed to the calling function that continues the processing. Functions can call themselves. Due to this recursion, local variables, assignments, or references of any kind to intermediate storage are no longer needed. The state of the program is given at any time exactly by the current expression. Hence, there are no side effects, such as clashes between local and global variable identifiers. A new piece of program can be simply added by inserting an additional function that is applied to the current expression. Usually the functions are so-called "pure functions." They are not assigned specific names; the name of a pure function is the encoding character string itself.

The language, referred to as AlChemy (a shorthand for Algorithmic Chemistry), is extremely simple. A detailed definition of AlChemy is given in the appendices. The following paragraphs give a qualitative overview of syntax and semantics.

3.1 SYNTAX

The program strings of AlChemy are combinations of characters taken from a set $\mathcal{C}$ that includes right and left parentheses. All characters except the parentheses are called atoms.

A syntactically legal string must fulfill only one requirement: the number of left and right parentheses must balance for the first time at the end of the string.

Syntactically legal strings are called expressions. According to the above definition a single atom is an expression. The parentheses, zero in this case, balance after the single character. Expressions consisting of more than one atom must, therefore, begin with a left parenthesis and must terminate with a right parenthesis. Such an expression is often called a list. Inside a list, parentheses can group atoms together.

Groups delimited by matching parentheses are obviously expressions. Hence, expressions can consist of other expressions.

The definition of an expression is precisely the recursive definition of a tree. Every pair of matching parentheses is represented by an internal node, while every atom is represented by a terminal node or leaf. Figure 1 gives a graph interpretation of expressions as rooted, ordered trees. Ordered means that the relative order of the subtrees is important. The empty list, (), is treated like an atom, but consists of two characters (Figure 1). The set of legal objects in the universe of AlChemy is therefore the set of all trees. The parentheses are purely structural characters needed to encode a tree graph as a linear string that can be manipulated by a computer in a convenient way.

The length of an expression is the number of characters in that expression. The number of expressions of length n, E_n, is derived in Chaitin.[4] The asymptotic estimate, $E_n = |\mathcal{C}|^{-1/2}\,|\mathcal{C}|^n/(2n\sqrt{\pi n})$, with $|\mathcal{C}|$ denoting the cardinality of $\mathcal{C}$, equals almost the number of strings of length n, $|\mathcal{C}|^n$. This indicates that the syntactic constraints are rather weak, and they become more so as n increases.

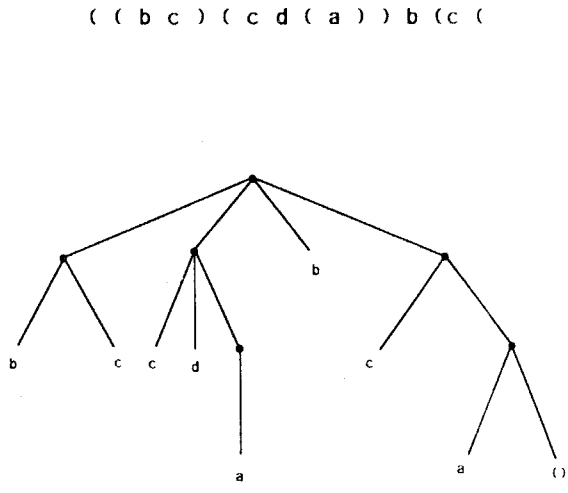


FIGURE 1 Expressions and trees. As in LISP, every AlChemy expression (top) can be represented as an ordered tree (bottom). A pre-order traversal reconstructs the list expression.

3.2 SEMANTICS

The semantics of a language determines what an expression "means." Expressions shall be used to represent programs, i.e., specifications of symbolic algorithms. A program A accepts an expression as input and returns an expression as output. Let $\mathcal{T}$ be the set of all trees (expressions), and let $\mathcal{X}$ be the subset of $\mathcal{T}$ for which A terminates. The meaning of A is then precisely the function $f : \mathcal{X} \subset \mathcal{T} \mapsto \mathcal{Y} \subset \mathcal{T}$, where $x \in \mathcal{X}$ is the input and $f(x) \in \mathcal{Y}$ is the output, or value, at termination. Clearly, different expressions (programs) can "mean" the same function. In order to provide a semantics, a set of rules has to be defined that specifies how expressions can manipulate symbol strings, thus encoding a function.

When constructing a function, a character symbolizing the variable has to be specified. The functions considered in this contribution will have only one variable. Let the character denoting this variable be "a."

In functional programming, functions are defined in terms of other functions. This is reflected on the syntactic level by the recursive definition of expressions being made of other expressions. The atoms are the base step in this recursion. A similar base step holds on the semantic level. At some point the recursive definition of a function must terminate. Functions that are no longer defined in terms of other functions are termed "primitive operators," or "primitives." The action of primitive operators is defined as a part of the semantics of the language. Primitives are "hard-wired," predefined operations acting on some specified number of argument expressions. They are assigned specific atoms of the alphabet $\mathcal{C}$ as names.

Since primitives operate on expressions, their actions consist in basic manipulations of tree structures. For example, joining two trees or deleting subtrees. The overall semantics of the language is defined in such a way that operators can easily be added, or their definition changed. The number of operators, as well as their actions, set the level of description in the model class that is being described.

The set of operators used in this version of AlChemy is limited to only six very simple ones whose functions are defined in Appendix B. All operators return legal expressions.

In AlChemy, an expression denotes a function, $f(\mathtt{a})$, and the value that f assigns to a particular argument r is the "value" of the expression f computed when the variable a is replaced by the expression r. The basic process of "evaluation" is defined recursively, and is outlined in the next paragraph. This process can be viewed as assigning a value to the root of the expression tree in terms of the values of its children. When the value of an expression has been obtained, it always replaces that expression.

The base step is, therefore, to assign a value to an atom. A terminal node must be an atom, and, hence, be either the variable a or a primitive operator. Operators shall always evaluate to themselves: their symbols are never substituted. The operator $*$, for example, has always value $*$. The value of the variable a, in contrast, is looked up in a list called the "association list." The association list is a look-up table where an expression is assigned to the atom a. If this list has no entry for a, then a is not substituted.

Assigning a value to some internal node of the expression tree involves using the values of its children in left to right order. Since these values are expressions, they denote functions that are to be applied to their arguments. Recall that there are two types of functions: primitive operators and composite functions. Accordingly, the "application" of a function to one or more arguments is performed in two ways.

If the function is an atom denoting an n-ary primitive operator, n siblings following the operator node are evaluated and their values taken as arguments. The built-in operation corresponding to the primitive is then applied to the argument expressions. If an operator encounters an insufficient number of arguments, an empty list, ( ), is supplied for each missing expression. Any excess arguments are ignored.

If the function is composite, the right neighbor sibling is evaluated and its value taken to be the argument of the (one-variable) function. If there is no sibling, an empty list is supplied. The procedure amounts to update the association list of that function, such that the argument expression is assigned to the variable a. The expression denoting the function is now evaluated using the new association list. The value expression obtained in this way replaces the original expression.

Figure 2 illustrates the evaluation process. The expression ((+a)(−a)) is evaluated using the association list that assigns the value ((∗aa)(+a)) to the variable a. The interpretation process follows the tree structure until it reaches an atom. In this case it happens at depth 2. The atoms are evaluated: the operators "+" and "−" remain unchanged, while the value of a is looked up in the association list that assigns to it the value ((∗aa)(+a)). The interpreter backs up to compute the values of the nodes at the next higher level using the values of their children. The value of the left node at depth 1 is obtained by applying the unary "+"-operator to its sibling (which has already been evaluated). The "+" operation returns the first subtree of the argument, (∗aa) in this case. Similarly, the value at the right depth 1 node is obtained by applying the unary "−"-operator to its argument ((∗aa)(+a)). The "−" operation deletes the first subtree of its argument returning the remainder, (+a). The interpreter now has to assign a value to the top node. The left child's value is a non-atomic expression and, therefore, denotes a composite function. This function is (∗aa), labelled as f in Figure 2. Its argument is the right neighbor sibling, (+a), labelled as g. Evaluating the top node means applying f to g. This is done by evaluating f with an association list that assign to a in f the value g. The procedure then recurs along a similar path as above, shown in the box of Figure 2. The result of $f(g)$ is the expression ((+a)(+a)). This is the value of the root (level 0) of the original expression tree, and therefore the value of the whole expression, given the initial association list.

What happens if a node has more than two children whose values represent composite functions, as in (f g h)? In AlChemy the value of this expression is defined to be ($f(g)$ $g(h)$): every function is applied to its right neighbor in turn and the results are appended to the value expression. If the application results in an empty list ( ), denoting the NIL function, the empty list is not appended. The last

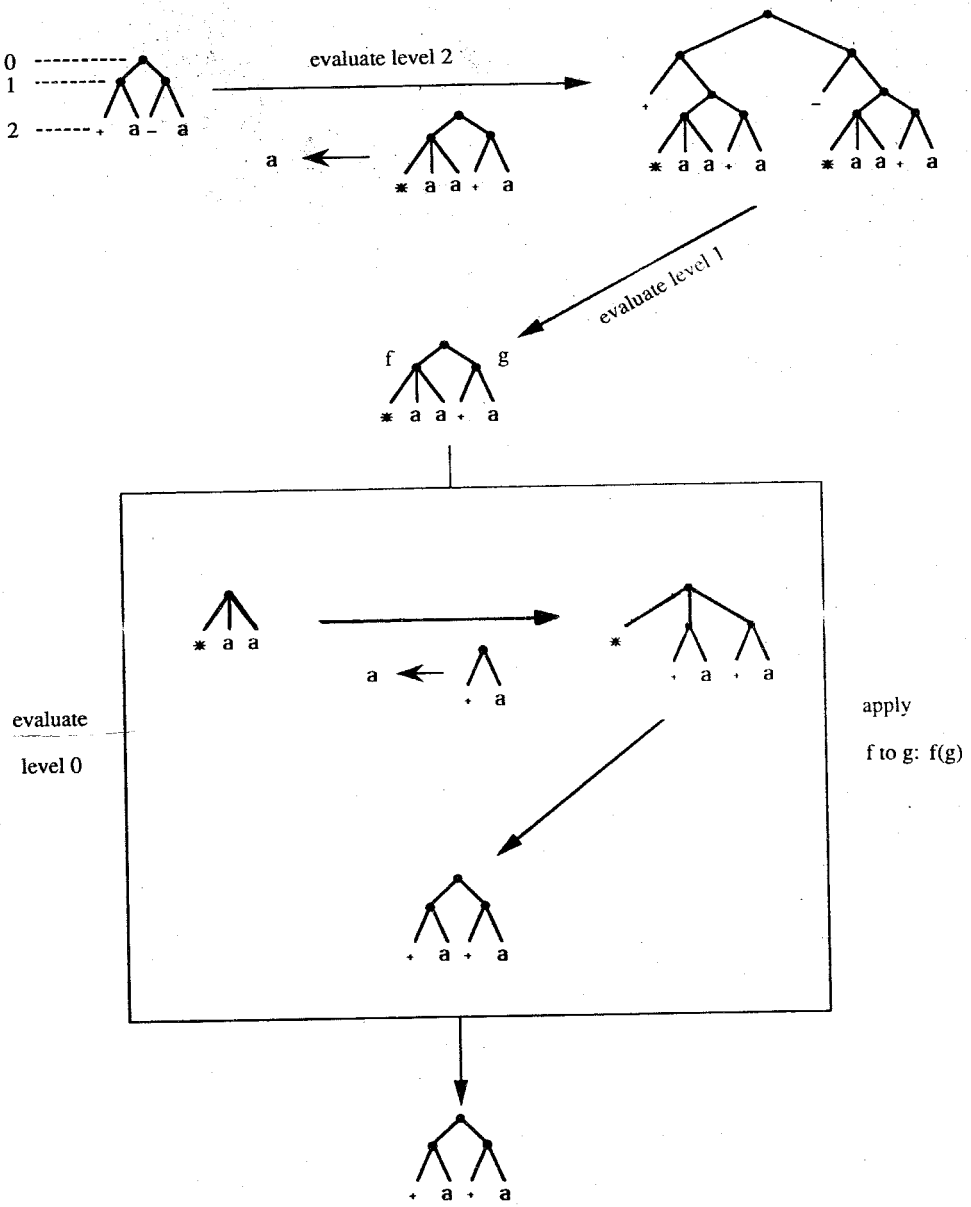


FIGURE 2 Evaluation example. The expression $((+\,a)(-\,a))$ is evaluated using an association list that assigns the expression $((*\,a\,a)(+\,a))$ to the variable a. See text for details.

function h would be applied to the empty list supplied for the missing argument. This almost always results in the NIL function. The evaluation is, therefore, skipped *a priori*.

The resolution of such a situation is not standard. The value $(f(g(h)))$ would be more in the spirit of functional programming. However, many computer experiments clearly indicate that with the current set of operators a long chaining of functions evaluates too frequently to NIL. This is partly due to the fact that the majority of the current operators shorten expressions. The sequential scheme is a simple action against this effect.

The language outlined here has a few minor idiosyncrasies relative to toy-LISP as defined in Chaitin.[4] For more details see the appendices. A long series of experiments with variations in the semantics has been performed. All experiments gave results whose basic structure was essentially identical to those obtained with this version and reported in section 6.

3.3 INTERACTION BETWEEN FUNCTIONS

The hierarchical structure of functions implied a recursive evaluation process that relies on the application of subfunctions to other subfunctions. This results immediately in a natural definition of interaction.

Let f and g be expressions. The natural way to let them interact is to construct an expression whose value is $f(g)$ or $g(f)$, depending on who has been chosen to act on whom.

According to the semantics of section 3.2, the expression $(\,f\;g\,)$ is not suited, because its value is obtained by first evaluating the expressions f and g separately and then applying the value of f to the value of g. In order to avoid the evaluation of an expression, a primitive operator is defined whose action is to return the unevaluated argument expression. This operator is denoted by the symbol $'$, and is referred to as the "quote"-operator.

The correct "interaction expression" then must read

$$((\,'f)('g))\,, \tag{9}$$

and is evaluated using an initially empty association list (a assigned to a). The values of $('f)$ and $('g)$ are f and g, respectively. The interaction expression then evaluates to $(f(g))$, which is the value that has been sought (see Figure 3).

Notice that the term "interaction" is here just the name of a syntactic expression with the particular structure (9), and is thus itself an object belonging to the language. More generally, let $\mathcal{F}$ be the set of functions defined by the language. "Interaction" is, then, the name of any two-variable function Φ—not necessarily composition—that assigns to any ordered pair of functions (f,g) some function h:

$$\Phi : \mathcal{F} \times \mathcal{F} \mapsto \mathcal{F},\ (f,g) \to h\,. \tag{10}$$

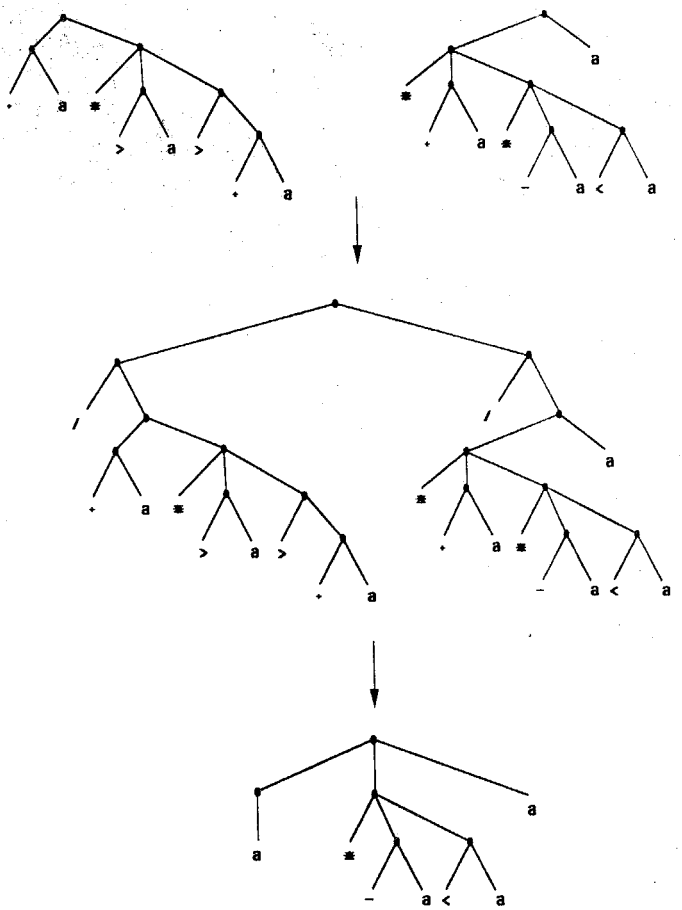


FIGURE 3 Interaction between algorithms. Two algorithmic strings (top) represented as trees interact by forming a new algorithmic string (middle) that corresponds to a function composition. The new root with its two branches and '-operators is the algorithmic notation for composing the functions. The interaction expression is evaluated according to the semantics of the language and produces an expression (bottom) that represents a new function. The evaluation of the interaction expression is derived in Appendix C.

The expression resulting from any particular Φ after insertion of particular expressions for f and g is clearly a function in one variable; see Figures 2 and 3. Nevertheless, on a meta-level, the set of possible interaction forms is the set of two-variable functions, of which Eq. (9), $\Phi(\mathtt{a},\mathtt{b}) = ((\,'\mathtt{a}\,)(\,'\mathtt{b}\,))$, is but one, albeit natural, example. For reasons of simplicity the present language has been restricted

to functions in one variable. The interaction form, though having two variables, obviously operates according to the semantics. When the number of variables is not limited, Φ is always an element of $\mathcal{F}$.

The same semantics that determines the function meant by any individual expression determines in particular the meaning of "interaction," for example, Eq. (9). The unity of function and interaction is a fundamental feature resulting from the completeness of this model class.

Figure 3 depicts the current interaction scheme. The evaluation of that particular interaction is derived step by step in Appendix C, which also serves as a more detailed interpretation example.

3.4 REMARKS

The above-described language is an extremely simplified version. The primitive operators (Appendix B) are not very powerful. More powerful primitive function constructors, as suggested for example in Arbib,[22] can be used.

The system allows sequences of primitive operations to be nested and grouped together into an expression. The value of such an expression is a function. In the present version a function acts precisely like a unary operator. It is, in contrast to a primitive operator, not hard-wired. It can be decomposed, reshuffled, or joined to something else. By means of interactions among individual functions, the system can, therefore, construct further "composite" operators. In addition, the system has not to provide names for its newly constructed operators. As in the λ-calculus, the specific sequence of characters coding for a function represents its name. There is no limitation, in principle, to the number of composite operators that a system can sustain.

The present semantics considers only functions in one variable. The extension to n variables is straightforward. A universe of multivariable functions leads to interesting additional questions due to n-"body" interactions. This theme is resumed in section 8.

The recursiveness of the evaluation process bears the danger of infinite loops. In Figure 2 the expression $((*\mathtt{a}\,\mathtt{a})(+\mathtt{a}))$ is assigned to $\mathtt{a}$ at the beginning of the evaluation process. The reader can verify that replacing the $+$ operator with $\mathtt{a}$ results in an infinite loop during the evaluation process shown inside the box of Figure 2. This is avoided by allocating to each interaction a depth limit. The depth limit specifies how many nested function evaluations are allowed to be incomplete at any given time. An evaluation that exceeds the depth limit stops gently by simply returning the value expression that has been computed so far (wrapped properly in parentheses). To avoid long cyclings between evaluation tree levels, an additional limitation had to be introduced: an interaction has to be completed within some maximum real cpu time. Similarly, only a limited amount of memory space is allowed for evaluating an interaction.

The interpreter is written in C, and is available from the author upon request.

4. ITERATED INTERACTION GRAPHS

The interactions between functions in a set $\mathcal{A}$ can be represented as a directed graph G. A graph G is defined by a set $V(G)$ of vertices, a set $E(G)$ of edges, and a relation of incidence, which associates with each edge two vertices (i, j). A directed graph, or digraph, has a direction associated with each edge. A labelled graph has, in addition, a label k assigned to each edge (i, j). The labelled edge is denoted by (i, j, k).

The action of function $k \in \mathcal{A}$ on function $i \in \mathcal{A}$ resulting in function $j \in \mathcal{A}$ is represented by a directed labelled edge (i, j, k):

$$(i, j, k): \quad i\, k \rightarrow j \quad i, j, k \in \mathcal{A} \tag{11}$$

Note that the labels k are in $\mathcal{A}$. The relationships among functions in a set are then described by a graph G with vertex set $V(G) = \mathcal{A}$ and edge set $E(G) = \{(i, j, k) : j = k(i)\}$.

A useful alternative representation of an interaction is in terms of a "double-edge,"

$$(i, j, k): \quad i\,(i, k) \longrightarrow j\,(i, k) \longleftarrow k \quad i, j, k \in \mathcal{A}, \tag{12}$$

where the function k acting on i and producing j has now been connected to j by an additional directed edge. The edges are still labelled, but no longer with an element of the vertex set. The labels (i, k) are required to uniquely reconstruct the edge set from a drawing of the graph. The graph corresponding to a given edge set is obviously uniquely specified. Suppose, however, that a function j is produced by two different interactions. The corresponding vertex j in the graph then has four inward edges. Uniquely reconstructing the edge set, or modifying the graph, for example by deleting a vertex, requires information about which pair of edges results from the same interaction. Some properties of the interaction graph can be obtained while ignoring the information provided by the edge labels. The representation in terms of double edges (i, j, k) has the advantage to be meaningful for any interaction function Φ mapping a pair of functions (i, k) to j, and not only for the particular Φ representing chaining. The double-edge suggests that both i as well as k are needed to produce j. In addition, the asymmetry of the interaction is relegated to the label: (i, k) implies an interaction $\Phi(i, k)$ as opposed to $\Phi(k, i)$. This representation is naturally extendable to n-ary interactions $\Phi(i_1, i_2, \ldots, i_n)$. In the binary case considered here every node in G must therefore have zero or an even number of incoming edges.

The following gives a precise definition of an interaction graph G. As in Eq. (8) let $\Phi[\mathcal{A}]$ denote the set of functions obtained by all possible pair collisions $\Phi(i, k)$ in $\mathcal{A}$, $\Phi[\mathcal{A}] = \{j : j = \Phi(i, k), (i, k) \in \mathcal{A} \times \mathcal{A}\}$. The interaction graph G of set $\mathcal{A}$ is defined by the vertex set

$$V(G) = \mathcal{A} \cup \Phi[\mathcal{A}] \tag{13}$$

and the edge set

$$E(G) = \{(i,j,k) : i,k \in \mathcal{A}, j = \Phi(i,k)\}. \tag{14}$$

The graph G is a function of $\mathcal{A}$ and Φ, $G[\mathcal{A},\Phi]$. The action of the map

$$M : \mathcal{A}_{i+1} = \Phi[\mathcal{A}_i] \tag{15}$$

on a vertex set $\mathcal{A}_i$ leads to a graph representation of M. Let

$$G^{(i)}[\mathcal{A},\Phi] := G[\Phi^i[\mathcal{A}],\Phi] \tag{16}$$

denote the ith iteration of the graph G starting with vertex set $\mathcal{A}$; $G^{(0)} = G$.

A graph G and its vertex set $V(G)$ are closed with respect to interaction, when

$$\Phi[V(G)] \subseteq V(G); \tag{17}$$

otherwise, G and $V(G)$ are termed innovative.

Consider again the map M, Eq. (15). What are the fixed points of $\Phi[\cdot]$? $\mathcal{A} = \Phi[\mathcal{A}]$ is equivalent to (a) $\mathcal{A}$ is closed with respect to interaction, and (b) the set $\mathcal{A}$ reproduces itself under interaction. That is,

$$\forall j \in \mathcal{A}, \exists\, i,k \in \mathcal{A} \text{ such that } j = \Phi(i,k). \tag{18}$$

Condition (18) states that all vertices of the interaction graph G have at least one inward edge (in fact, two or any even number). Such a self-maintaining set will also be termed "autocatalytic," following M. Eigen[9] and S. A. Kauffman[17,18] who recognized the relevance of such sets with respect to the self-organization of biological macromolecules.

Consider a set $\mathcal{F}_i$ for which Eq. (18) is still valid, but which is not closed with respect to interaction. $\mathcal{F}_{i+1}$ obviously contains $\mathcal{F}_i$, because of Eq. (18), and in addition it contains the set of new interaction products $\Phi[\mathcal{F}_i] \setminus \mathcal{F}_i$. These are clearly generated by interactions within $\mathcal{F}_i \in \Phi[\mathcal{F}_i]$. Therefore, Eq. (18) also holds for the set $\Phi[\mathcal{F}_i]$, implying that the set $\mathcal{F}_{i+1}$ is autocatalytic. Therefore, if $\mathcal{A}$ is autocatalytic, it follows that

$$G[\mathcal{A},\Phi] \subseteq G^{(1)}[\mathcal{A},\Phi] \subseteq G^{(2)}[\mathcal{A},\Phi] \subseteq \ldots \subseteq G^{(i)}[\mathcal{A},\Phi] \subseteq \ldots. \tag{19}$$

In the case of strict inclusion, let such a set be termed "autocatalytically self-extending." Such a set is a special case of innovation, in which

$$\Phi[V(G)] \supseteq V(G) \tag{20}$$

holds, with equality applying only at closure of the set.

An interesting concept arises in the context of finite, closed graphs. Consider, for example, the autocatalytic graph G in Figure 3(b), and assume that G is closed.

The autocatalytic subset of vertices $V_1 = \{A, B, D\}$ induces an interaction graph $G_1[V_1, \Phi]$. Clearly, $G[V, \Phi] = G_1^{(2)}[V_1, \Phi]$, which means that the autocatalytic set V_1 regenerates the set V in two iterations. This is not the case for the autocatalytic graph shown in Figure 3(a). More precisely, let G be a finite-interaction graph, and let $G_\alpha \subseteq G$ be termed a "seeding set" of G, if

$$\exists\, i, \text{ such that } G \subseteq G_\alpha^{(i)}, \tag{21}$$

where equality must hold if G is closed. Seeding sets turn out to be interesting for several reasons. For instance, in the next section a stochastic dynamics (Turing gas) will be induced over an interaction graph. If a system is described by a graph that contains a small seeding set, the system becomes less vulnerable to the accidental removal of functions. In particular cases a seeding set can even turn the set it seeds into a limit set of the process. Such a case arises when every individual function f_i in $\mathcal{A}$ is a seeding set of $\mathcal{A}$:

$$\begin{aligned} f_{i+1} &= \Phi(f_i, f_i), \qquad i = 1, 2, \ldots, n-1 \\ f_1 &= \Phi(f_n, f_n). \end{aligned} \tag{22}$$

Furthermore, suppose that G is finite, closed, and autocatalytic. It follows from the above that all seeding sets G_α must be autocatalytically self-extending, as for example in Figure 3(b). If G is finite, closed, but not autocatalytic, there can be no seeding set. Being closed and not autocatalytic implies $V(G^{(2)}) \subset V(G)$. The vertices of G that have no inward edges are lost irreversibly at each iteration. Therefore, for some i either $G^{(i)} = \emptyset$, or $G^{(i)}$ becomes an autocatalytic subset of G.

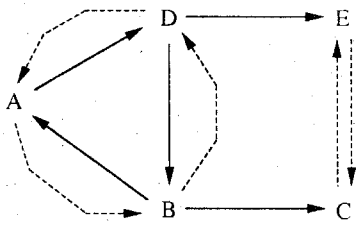
In the case of innovative, not autocatalytic sets, i.e., sets for which

$$\Phi[\mathcal{A}] \not\subseteq \mathcal{A} \wedge \Phi[\mathcal{A}] \not\supset \mathcal{A} \tag{23}$$

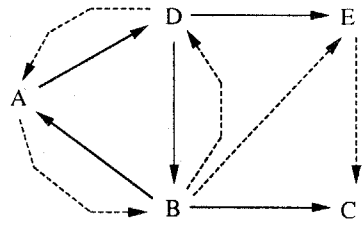
holds, no precise statement can be made at present.

A digraph is called connected if, for every pair of vertices i and j, there exists at least one directed path from i to j and at least one from j to i. An interaction graph G that is connected not only implies an autocatalytic vertex set, but in addition depicts a situation in which there are no "parasitic" subsets. A parasitic subset is a collection of vertices that has only incoming edges, like the single vertices C and E in Figure 3(b), or the set $\{C, E\}$ in Figure 3(a). As the name suggests, a parasitic subset is not cooperative, in the sense that it does not contribute to generate any functions outside of itself.

All the properties discussed in this section are independent of the information provided by the edge labels (in the double-edge representation). Note, furthermore, that the above discussion is independent of any particular representation of "function." It never refers to the implementation in the LISP-like AlChemy presented in section 3. The representation of function in terms of that particular language is used in the simulations of section 6 to demonstrate the accessibility of the phenomena outlined above, as well as to provide a workbench for experimentation.



(a)



(b)

FIGURE 4 Self-maintaining (autocatalytic) graphs. The lower graph, (b), can be regenerated from the vertex subset $\{A, B, D\}$, in contrast to the upper graph, (a). Both contain parasitic subsets: $\{E, C\}$ in (a), and $\{C\}$, $\{E\}$ in (b). See text for details.

5. A TURING GAS

The previous section briefly considered the dynamics of relationships among objects in a set as they interact with each other on the basis of a formal language. These relationships are captured at any time by a graph. In physical instances the nodes of the graph could represent currently available interactive entities, as molecules, species, instructions, messages, etc. Usually these entities can be present in multiple copies. The nodes of the graph then support a frequency distribution that induces an interaction kinetics on the graph. The change in frequency of a particular object will depend on the frequencies of those objects that are needed for its production, as in Eq. (5).

In this section a simple stochastic process is used to induce a dynamical system on the interaction graphs. The size of the system is kept finite and constant. The constant size clearly represents a selection constraint that will influence the systems' evolution.

Let the system contain N functions. An iteration consists of two steps: a random collision between two objects and the application of a scheme to keep the total number of objects constant. The collision step is as follows:

1. Collision:

   a. choose at random two objects f and g from the system, and

   b. evaluate their interaction expression. This generates a new expression h. If the expression h (1) contains at least one primitive operator, (2) contains at least one variable, and (3) is not longer than $l_{\max}$ characters, then the collision is termed "reactive," and the reaction products are f, g, and h:

$$f + g \rightarrow (('f)('g)) \rightarrow h + f + g. \tag{24}$$

   If any of the above conditions is not fulfilled, then the collision is termed "elastic":

$$f + g \rightarrow (('f)('g)) \rightarrow f + g. \tag{25}$$

2. Removal: If the collision was reactive one of the old N functions is chosen at random and deleted. The reaction product h is, therefore, kept in any case.

The first chosen object, f, operates on the second object g. No conservation of any quantity is imposed during the collision. This keeps the scheme simple at first. The language definition itself provides for some constraints. For example, the product h cannot contain any primitive operator that was not already present in f or g. Keeping the interacting functions f and g after a reactive collision is in line with their abstract nature. f and g represent information that is used to build a new object. Information is not destroyed by the mere fact of its usage. Nevertheless, f and g are subject to the dilution flux—a random erasure—that is necessary to establish a finite system.

$l_{\max} = 300$ in all examples discussed in section 6. In addition, to avoid halting problems the computation of a collision has not to exceed a depth of 10 (see section 3.4) and has to be completed within 6 seconds of real cpu time.

The scheme of constraining the number of particles is essentially equivalent to a flow reactor. The encounter between two object "species," as well as their dilution due to removal, occurs with a probability proportional to their frequency in the system.

The system is typically started with N random functions. A random function is a syntactically legal random string of characters obtained as outlined next.

In what follows, matching parentheses are always wrapped around an n-ary operator and its n-argument expressions. Generating an expression is then a very simple recursion R: (1) an atom is drawn at random; (2) if the atom is the variable, a complete (atomic) expression has been obtained and the procedure stops, else the atom is an n-ary operator that requires n-argument expressions that are generated according to procedure R.

The random-function generator has two parameters. One parameter concerns the probability by which an operator character is chosen. This allows to tune the frequency of operators versus variable. The above procedure, however, generates only trees with at most one primitive operator plus corresponding argument expressions

at each level. Expressions could consist of any finite number k of branchings at any internal node. The second parameter tunes k, but only for the first level. To build an expression the above generator is simply invoked k times, and a pair of parentheses is wrapped around the k expressions.

The above procedure can access only a subset of all functions, in particular those that have operators associated with the "correct" predefined number of arguments. It seems natural to start with such functions, although any tree would be legal.

6. RESULTS AND DISCUSSION

In this section I discuss some of the basic results obtained in the first experiments with the Turing gas. Three variants will be considered. The first is a "plain" version, where the system self-organizes from random initial conditions into "quasi-stationary" (see below) interaction patterns. The second version slightly modifies the collision scheme such as to forbid copy reactions that are basic to the interaction patterns that develop in the plain version. Both versions keep the system closed. The third version opens the system at a quasi-stationary state and periodically perturbs the system by releasing a small percentage of new random functions into the gas.

6.1 TURING GAS WITHOUT PERTURBATIONS

The gas starts with $N = 1000$ randomly generated functions, each present in a single copy. Table 1 gives a glimpse of an initial condition. The random-function generator was instructed to generate function expressions containing four branches at the first level. Each expression was produced by drawing an operator with probability 0.7 and a variable with probability 0.3. Table 2 lists the state of the gas after 3×10^5 collisions, 93264 of which have been reactive. The 1000 particles are now distributed over only 18 different functions listed in lexicographic order with their corresponding number of copies. All functions are different from those present initially. Figure 5 shows the interaction graph G (section 4) obtained by performing all 324 pairwise collisions. The origin of a dotted line connecting to a solid arrow indicates the function that transforms the tail of the arrow into the head. This representation deviates slightly from the definitions in section 4, but has been chosen for the sake of clearer and less congested figures. Functions are usually "named" by their lexicographic order in the set under discussion. Capital letters denote sets of functions. Sometimes several syntactically related functions perform similar operations on different but again syntactically related arguments. This can be conveniently represented by having arrows and dotted lines connect sets instead of single functions. A set C transforming a set A into a set B then means

$$\forall j \in B \; \exists! \; i \in C, k \in A \text{ such that } j = i(k). \tag{26}$$

Inspection of the pair interaction data shows that function 17 is an identity function, or "general copier." An identity function is a function f with $f(g) = g, \forall g$. In some simulation experiments several identity functions are produced that copy themselves and each other.

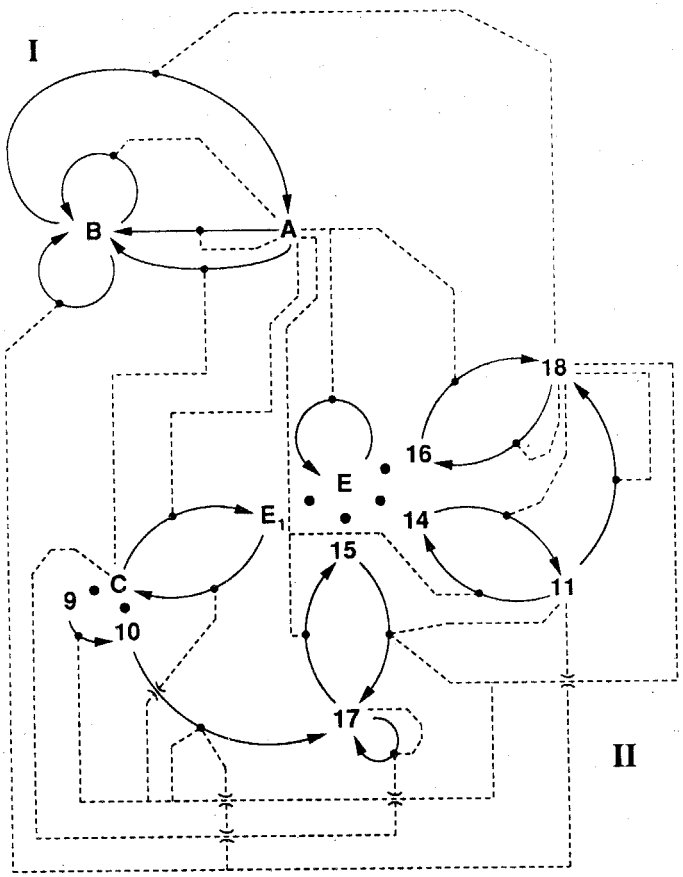


FIGURE 5 Interaction graph. The interaction graph of the functions listed in Table 2 is shown. The numbers denote the individual functions according to their ordering in Table 2. Capital letters denote sets, where $A = \{1,2,3,4\}$, $B = \{5,6,7,8\}$, $C = \{9,10\}$, $E = \{12,13,14,15,16\}$, and $E_1 = \{12,13\} \in E$. Solid arrows indicate transformations, dotted lines functional couplings. A dotted line originates in a function (or a set, see text), say, k, and connects (filled circle) to a solid arrow, whose head is j and whose tail is i. This is to be interpreted as $j = k(i)$. Large filled circles indicate membership in a particular set. Function 17 is an identity function. Note: all dotted lines and solid arrows that result from 17 copying everything else in addition to itself have been omitted. See text for details.

Note that, in a universe defined by a language, there are two senses in which things can be equal. One sense refers to the semantical level: two objects f and g are equal if they represent the same function, i.e., if (1) they have the same domain of definition $\mathcal{D}$ ($\mathcal{D}$ is the set of arguments h for which $f(h)$ terminates), and if (2) $f(h) = g(h), \forall h \in \mathcal{D}$. The other sense refers to the syntactical level: two objects are equal if their symbol strings are equal. Throughout this paper equality refers to the syntactical level. Several objects may represent the identity function, but differ in their symbolic representation. The identity function, for example, can be constructed in multiple ways by encoding first a series of operations to be performed on an input string, and then a second series that "undoes" the first one. Semantical (functional) equality can be very hard to establish. In fact, a general procedure for establishing functional equivalence between any two objects would run into the halting problem.

The drawing of the interaction graph in Figure 5 is incomplete, because it should contain for each function one solid self-loop with a dotted line coming from the general copier 17. These interactions were left out for the sake of a clearer picture. The reader is asked to keep in mind that all functions in Figure 5 are copied by 17, not only 17 itself.

TABLE 1 Random functions. The table shows randomly generated functions containing four expressions at the first tree level. The probability of drawing an operator was 0.7, the probability of chosing a variable was 0.3. The parentheses are completely determined by the sequence of operators. Thus only a subset of legal expressions is generated: expressions whose operators are bound to a complete set of arguments.

```
((>a)a(>(*aa))a)
((>('a))(*aa)(-a)a)
((<('a))(-(*a(-(<(+a)))))(<('(-(<a))))(-a))
((-(*(*a(<(<(<(>a)))))a))aaa)
(('('('(+a))))a('(-(*(+('(>('a))))(*a(>a)))))(*(<a)a))
((+a)a('a)(-(>(+(*(>(+(*(<a)(-a))))(>a))))))
(('('('(-a))))a(-(>a))(>a))
((*(*aa)a)(*(-(<(<(-a))))a)(>(*(*(-(-(+(>('a)))))(-a))(-('a))))(<a))
((*(*(>(-a))a)(<(<(-a))))aaa)
(('('(*(+a)(>(+(-('(+a)))))))(+(+a))(+(<(<(-(<a)))))a)
(a(-a)('('(*(-a)a)))(<('(<(>a)))))
(aa(-(+(*(+(<a))('a))))(+(+(-a))))
```

TABLE 2 State of an unperturbed Turing gas. The table lists the state of a Turing gas with $N = 1000$ particles after 3×10^5 collisions. First column: lexicographic order of the function. This number is the "name" used in the text to refer to a particular function. Second column: – marks indicate functions that disappear during the following 2×10^5 collisions. Third column: number of copies. Fourth column: function expression. See Figure 5 for the interaction graph and text for the details.

(# 1)	–	14	(('a)(a))('a)(a))
(# 2)		43	((('a)(a))('a)(a))(('a)(a))('a)(a))
(# 3)	–	16	(((((>('(>a)))(>('(>a))))(a))((>('(>a)))(>('(>a))))(a))(((>('(>a)))(>('(>a))))(a))((>('(>a)))(>('(>a))))(a))
(# 4)		90	((((>('(>a)))(>('(>a))))(a))((>('(>a)))(>('(>a))))(a))
(# 5)		33	(((a)(a))(('a)(a))('a)(a)))
(# 6)		133	(((a)(a))((('a)(a))('a)(a))(('a)(a))('a)(a)))
(# 7)	–	36	(((a)(a))(((((>('(>a)))(>('(>a))))(a))((>('(>a)))(>('(>a))))(a))(((>('(>a)))(>('(>a))))(a))((>('(>a)))(>('(>a))))(a)))
(# 8)		205	(((a)(a))((((>('(>a)))(>('(>a))))(a))((>('(>a)))(>('(>a))))(a)))
(# 9)	–	10	((*a)((*a)(*a)))
(# 10)		7	((*a)(*a))
(# 11)	–	1	((>a)(>a))
(# 12)	–	37	((a)((*a)((*a)(*a))))
(# 13)		5	((a)((*a)(*a)))
(# 14)		7	((a)((>a)(>a)))
(# 15)		40	((a)(*a))
(# 16)		220	((a)(>a))
(# 17)		11	(*a)
(# 18)		92	(>a)

The component analysis[30] of the interaction graph G shows that G is connected. This implies that G is (1) self-reproducing (autocatalytic), (2) closed, and (3) has no parasitic subsets.

The constant system size represents a simple selective constraint. The only way for a particular function to survive in the system, in the long run, consists in becoming the product of some transformation pathway. The fate of that function is then linked to the functions in that pathway. To survive a pathway has to become closed, that is, self-maintaining. One (trivial) solution consists in a function that copies itself. Other solutions consist in sets that reproduce themselves without any single member being self-reproducing,[1,11,17] and any combinations of self-reproducing sets

and/or single functions. These solutions are precisely the fixed points of the iterated interaction map, Eq. (15). The stability of such self-reproducing sets is strongly influenced by the number and size of constituent seeding sets (section 4). Stochastic fluctuations continously wipe out functions. The smaller the minimal seeding set of a self-reproducing set, the higher its stability, because correspondingly large numbers of different functions that have been lost can be regenerated.

The Turing gas is a stochastic process whose fluctuations eventually drive the system into three types of absorbing barriers. (1) A possibly heterogeneous mixture of elastic colliders ("dead system"), (2) a single self-reproducing function, or (3) a self-reproducing set in which every single function is a seeding set. The latter is a subtle "steady state," although a rather contrieved one, since seeding sets are usually much bigger than a single function. In the unperturbed version most interesting situations are, therefore, confined to transients, in particular long-living transients. Such long-living transients will be called "quasi-stationary" states.

At 3×10^5 collisions the interaction graph G is closed. Initially all reactive collisions result in a new function. The fraction of innovative collisions (relative to reactive encounters) decays very fast during the first 30,000 collisions from a value of 1.0 to values fluctuating between 0.05 and 0.2. This range is kept for 80,000 further collisions, and drops, then, to zero as the system attains closure. It is important to distinguish between collisions that are innovative in the sense that the produced function is not present in the system at the time of its production, and collisions that produce functions that have never been in the system during its entire history. Collisions of the latter type are termed "absolutely" innovative (they are included in the count of innovative collisions). Fluctuations that wipe out lowly populated functions that are subsequently regenerated, or any retracing of past trajectories in function space, give rise to innovative collisions that generate functions that the system has already seen. In fact, in the present experiment the fraction of absolutely innovative collisions follows the decay of innovative collisions, but settles on values between 0.02 and 0.1 before eventually dropping to zero. A similar scheme—fast decay to a ratio of approximately 0.5 to 0.3 of absolute innovation versus total innovation for variably long periods of time—is observed in many computer experiments. The quasi-stationary state at termination of some simulations is closed with respect to interaction, while others (section 6.2) continue to be innovative. So far, none have been observed to remain absolutely innovative for over a million collisions. Nevertheless, initial conditions for which this may happen cannot be excluded in principle. Dependencies on the system size have not yet been systematically investigated.

The trend towards closure is based on the appearance of identity functions and partial copiers. The latter are functions that copy some but not all arguments. As soon as an identity function becomes the end product of a pathway, the members of that pathway will be generated in an autocatalytic fashion, since they are copied by their joint end product. These pathways subsequently extend themselves through innovative collisions (section 4). Functions not linked to these pathways are eventually displaced by dilution.

The graph G (Figure 5) exhibits two groups, I and II, of functions that are not connected with each other by solid arrows. Any function of a group can be transformed only into functions belonging to the same group. The groups are connected solely by functional edges (dotted lines): objects in one group operate some interconversions between objects of the other group.

If all connections between both groups were cut, for example by introducing a boundary, group II would still remain autocatalytic (in fact, connected) due to the action of the identity function 17 belonging to group II. Group I would eventually attain a state of pure elastic colliders belonging to set B. If Figure 5 were taken literally, i.e., if function 17 were not an identity but a pure self-replicator, the couple $\{15, 17\}$ would act as a parasite to the system. Its removal would leave the system connected. "Removal experiments" on an interaction graph can easily be performed by deleting particular nodes along with all those edges having them as tail or as head. For example, removing all nodes except $\{9, 10, 11, 15, 17\}$ shows that this subset is still connected, implying autocatalysis. A given self-maintaining set can be composed of several other self-maintaining sets: in Figure 5, for example, the whole system is self-maintaining, group II stand alone is self-maintaining, the subset $\{9, 10, 11, 15, 17\}$ is self-maintaining, and obviously function 17 is by itself. The nesting of autocatalytic components is a frequently observed pattern in the Turing gas. Connectivity of the whole set, and thus closure, is more readily attained in the presence of general copiers.

The interaction graph G', 10^5 collisions earlier, consisted of 39 functions forming a self-reproducing, but innovative set. The graph G is already contained in G'. G' is not connected, and therefore contains parasitic functions. In 10^5 collisions G' has been reduced to G.

Most of the computer experiments exhibit very complicated short-lived states that reduce to simpler cooperative metastable transients as in Figure 5. Figure 6 shows, as another example, the interaction graph of a different 1000 particle gas after half a million collisions. The corresponding functions are listed in Table 3. This example exhibits "partial copiers," i.e., functions that copy some, but not all, functions in the system. Depending on the argument function other transformations are performed. In this case, functions 1 and 4 are self-replicators. In addition, 1 also copies 4, but 4 does not copy 1. Instead, 4 copies the function 3 that is transformed back into 1 under the action of 3 itself and, of 1. 4, cooperates through this pathway indirectly with 1. The graph is innovative; the function denoted by a star is not in the basis set $\{1, 2, 3, 4\}$.

Partial copiers are an obvious example of how an interaction not only depends on the acting function, but also on the properties of the argument. The acting function can instruct some parts of the argument function to operate on other parts of itself. The fact that given some acting function particular reactions need particular arguments is analogous to recognition phenomena in molecular systems.

TABLE 3 State of an unperturbed Turing gas. The table lists the functions of a quasi-stationary state obtained in a Turing gas with $N = 1000$ particles after 5×10^5 collisions. The initial set of random functions was different than in the simulation of Table 2. See text for details.

(# 1)	785	((>(*(*('(>a))(+(+('a)))a))(*(>(-a))(>(*(*('(>a))(+(+('a)))a))))
(# 2)	4	((a)(>(*(*('(>a))(+(+('a)))a)))
(# 3)	98	(*(>(-a))(>(*(*('(>a))(+(+('a)))a)))
(# 4)	113	(>(*(*('(>a))(+(+('a)))a))

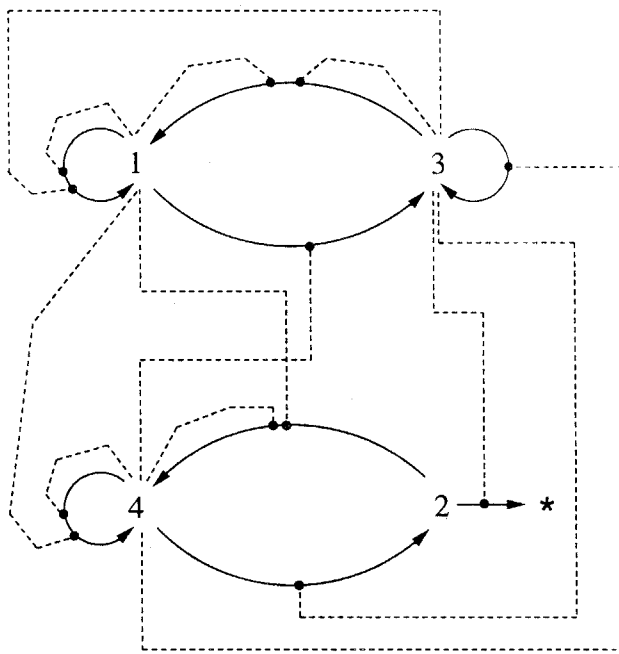


FIGURE 6 Interaction graph. The interaction graph of the functions listed in Table 3 is shown. See caption to Figure 5 and text for explanations.

6.2 TURING GAS WITHOUT COPY REACTIONS

The previous examples show that the organization of interactions in the Turing gas is centered at copy functions, be they general or partial. How does the system self-organize if copy reactions are not possible? This section considers this question as an example of a boundary condition imposed on the system through the collision rule.

The collision rule (section 5) is modified by simply declaring every copy reaction as being elastic. Note that AlChemy is left unchanged, and so is the definition of interaction.

One example only is discussed here as a prototype for many simulations. The 1000 different functions present initially have disappeared after 4×10^4 collisions. The similarity of two states at different times can be quickly assessed by computing the angle between the corresponding state vectors. The cosine of the angle between successive state vectors 5000 collisions apart from each other approaches values ranging between 0.97 and 0.99, as soon as the system reaches a quasi-stationary regime after approximately 2×10^5 collisions. At the same time, however, almost 50% of all the different function change constantly between each successive state. The innovation rate remains very high, but the absolute innovation rate becomes effectively zero after 3×10^5 collisions. The number of different functions fluctuates between 35 and 54. These facts indicate that the system steadily produces a fraction of new functions, loses them, and regenerates them again.

A snapshot of the gas after half a million collisions contains a set $\mathcal{A}$ of 51 different functions, the major part of which are too long and unwieldy to display. They are, however, built according to a simple scheme outlined below. The graph analysis reveals that the set is innovative and self-maintaining after removal of one function. This function has no inward edge, indicating again large fluctuations that led to the loss of its production pathway. Removing all edges pointing to innovative products, i.e., keeping only edges in the bulk $\mathcal{A}\backslash\Phi[\mathcal{A}]$, produces a graph that is connected, therefore self-maintaining and without parasites. How can a network be self-maintaining while constantly changing half of its functional species? The key to this consists in finding the minimal seeding set. The combinatorics makes this usually a difficult task. A simple heuristic that worked in the present case is to record a long series of n sets, $\mathcal{F}_i$, corresponding to states taken at particular time intervals, and to analyse their intersection set $\mathcal{I}$,

$$\mathcal{I} = \bigcap_{i=1}^{n} \mathcal{F}_i. \tag{27}$$

The intersection of 51 sets beginning at collision 2×10^5 and taken at intervals of 5000, contains 18 functions whose interaction graph is self-maintaining and innovative. The functions are listed in Table 4; their relationships are very simple and sketched in Figure 7.

TABLE 4 Turing gas without copy reactions. The table shows a quasi-stationary state of a Turing gas in which copy reactions were not allowed. The state consists of three polymer families A, B, and C, built from monomers $A1$, $B1$, and $C1$, respectively. The interaction graphs for subsets A and B are shown in Figure 7. The ordering of the functions is not lexicographic. See text for details.

(# A1)	41	(*(*aa)a)
(# A2)	69	((*(*aa)a)(*(*aa)a))
(# A3)	56	(((*(*aa)a)(*(*aa)a))(*(*aa)a))
(# A4)	20	((*(*aa)a)(((*(*aa)a)(*(*aa)a))(*(*aa)a))((*(*aa)a)(*(*aa)a)))
(# A5)	4	((((*(*aa)a)(*(*aa)a))(*(*aa)a)(*(*aa)a))(*(*aa)a)(*(*aa)a))
(# A6)	19	(((((*(*aa)a)(*(*aa)a))(*(*aa)a))((*(*aa)a)(*(*aa)a))(*(*aa)a))((*(*aa)a)(*(*aa)a))(*(*aa)a))
(# B1)	183	(<(*(>(*aa))(+a)))
(# B2)	115	((<(*(>(*aa))(+a)))(<(*(>(*aa))(+a))))
(# B3)	35	((<(*(>(*aa))(+a)))((<(*(>(*aa))(+a)))(<(*(>(*aa))(+a)))))
(# B4)	22	(((<(*(>(*aa))(+a)))(<(*(>(*aa))(+a))))(<(*(>(*aa))(+a))))
(# B5)	9	((<(*(>(*aa))(+a)))((<(*(>(*aa))(+a)))(<(*(>(*aa))(+a))))(<(*(>(*aa))(+a))))
(# B6)	13	((((<(*(>(*aa))(+a)))(<(*(>(*aa))(+a))))(<(*(>(*aa))(+a)))(<(*(>(*aa))(+a))))(<(*(>(*aa))(+a)))(<(*(>(*aa))(+a))))
(# C1)	182	(*(<(*(>(*aa))(+a))))
(# C2)	108	((*(<(*(>(*aa))(+a))))(*(<(*(>(*aa))(+a)))))
(# C3)	27	((*(<(*(>(*aa))(+a))))((*(<(*(>(*aa))(+a))))(*(<(*(>(*aa))(+a))))))
(# C4)	28	(((*(<(*(>(*aa))(+a))))(*(<(*(>(*aa))(+a)))))(*(<(*(>(*aa))(+a)))))
(# C5)	3	((*(<(*(>(*aa))(+a))))((*(<(*(>(*aa))(+a))))(*(<(*(>(*aa))(+a)))))(*(<(*(>(*aa))(+a))))))
(# C6)	8	((((*(<(*(>(*aa))(+a))))(*(<(*(>(*aa))(+a)))))(*(<(*(>(*aa))(+a))))(*(<(*(>(*aa))(+a)))))(*(<(*(>(*aa))(+a))))(*(<(*(>(*aa))(+a)))))

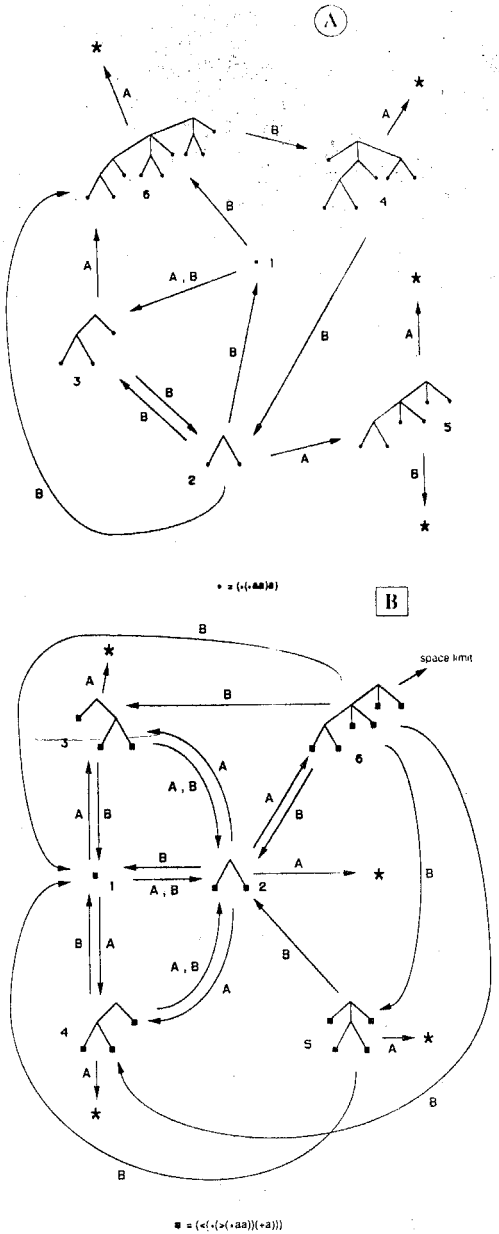


FIGURE 7 Interaction graph of a state without copy reactions. The interaction graphs of the A-subset, 7(a), and of the B-subset, 7(b), of Table 4 are shown. The functions are displayed as trees whose leaves are monomers given by the functional group indicated at the bottom of each graph. Solid arrows indicate transformations operated by some function(s) belonging to the subset(s) indicated by the label(s) of the arrows. Numbers as in Table 4.

The basic observation is that the functions become "polymers," and come in three distinct groups $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$, each containing six functions. Each group is characterized by its own "monomeric" unit: $A1$, $B1$, and $C1$. The monomers $B1$ and $C1$ are degenerate with respect to function, though they are different at the

syntactical level. The corresponding functions in each group act in the same way. Group $\mathcal{C}$ is, therefore, skipped in the discussion (and in Figure 7), since every statement about $\mathcal{B}$ applies equally to $\mathcal{C}$.

The basic polymerizing unit is monomer $A1 = (*(*\mathtt{a}\,\mathtt{a})\,\mathtt{a})$, indicated by a filled circle in Figure 7(a). Evidently, $A1$ triplicates any input **a**. Figure 7 shows the transformation pathways among the species in each group. Solid arrows are marked by letters indicating only the originating group of the function(s) that operate(s) the transformation. The system does not contain a function that interconverts monomers. Transformations occur therefore only within group members. $A1$ polymerizes every species in the system, provided the product does not exceed the 300 character limit imposed by the collision rule. This is the case for the 114-character function $B6$ in Figure 7(b). All polymerizations in group A are operated by $A1$. In particular, $A1 \rightarrow A3 \rightarrow A6 \rightarrow *$. Figure 8 gives the structure of the $A6$ trimer. The polymers are highly branched and self-similar due to the iterated action of $A1$. The intriguing feature is that the polymers created in all groups (mostly by action of $A1$) are themselves functionally active such as to establish depolymerization pathways that achieve self-maintaining closure of the system. They also form further products that are not "iterated trimers."

Both groups, $\mathcal{A}$ and $\mathcal{B}$ (and $\mathcal{C}$), are functionally strongly coupled. As Figure 7 shows, $\mathcal{A}$ is dependent for closure upon $\mathcal{B}$ and conversely. The essential interdependencies among the groups are visualized in Figure 9. One seeding set consists of a transformation cycle of three functions in each group. The cycles mutually depend on each other and interconvert their respective monomers, dimers, and trimers. The dimer and trimer vertices are further polymerized into higher-order structures.

Ultimately, Figure 9 and, therefore, the whole system can be built up from the mere presence of monomer $A1$ and monomer $B1$. Everything else follows. The reader can easily check: $A1$ and $B1$ are the minimal seeding set. Its smallness explains the high stability of this system.

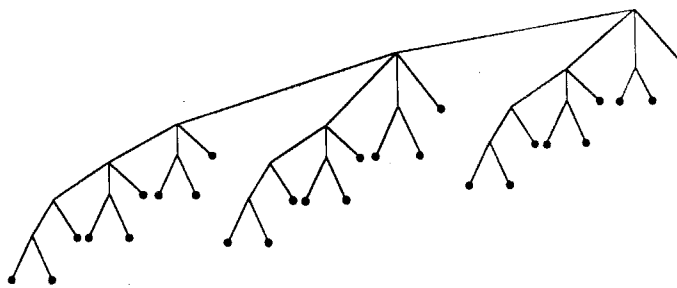


FIGURE 8 Self-similar polymer. The function resulting from three-fold application of the $A1 = (*(*\mathtt{a}\,\mathtt{a})\,\mathtt{a})$ monomer (see Table 4) to itself.

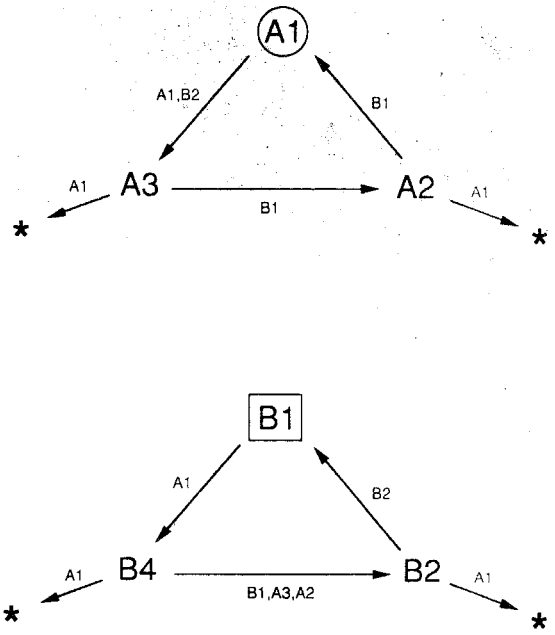


FIGURE 9 A seeding set and the minimal seeding set. A seeding set of the system shown in Figure 7. The two graphs correspond to the polymer families A and B. They represent the high populated core in Table 4 (a corresponding triangle must be added for group C). Arrows and labels as in Figure 7. The minimal seeding set of the system consists solely of the two monomers $A1$ and $B1$. See text for details.

The nine functions (including $\mathcal{C}$, not shown) of Figure 9 constitute the all-time-high populated core. The "polymerization triangles" cause a constantly high innovation rate. The system could expand indefinitely into new polymers, if it weren't for the length limitation on the function strings. In addition, the kinetics induced by the gas system and by the established pathways constrain the number of different function species that can be sustained with only 1000 particles. The ongoing polymerizations create a low-populated periphery subject to fluctuations that frequently lead to the loss of functions. The stable seeding set regenerates any lost structure. In fact, ten iterations of the map, Eq. 10, starting with $A1$ and $B1$ (and $C1$), generate a set that contains all the 51 functions constituting the system's state after 5×10^5 collisions.

Can seeding sets replace each other during the system's evolution? For example, a seeding set A generates some larger ensemble that happens to contain a different seeding set B for a different ensemble. B takes over replacing A. B now expands the new ensemble and hits a further seeding set C, and so on. In the example discussed so far, the system was trying to expand and to move away from a particular region in function space. However, the seeding set $\{A1, B1\}$ was never replaced, thus anchoring the system in that region. So far a replacement of seeding sets has not been observed in the unperturbed version of the Turing gas. It is observed regularly, however, when the gas is perturbed by exogenously introduced random functions (see section 6.3).

Some few experiments ended simply with elastic colliders. Most of the simulations, however, evolved organization patterns similar to those described above. All instances observed so far, produced different "polymeric" structures. Polymers consisting of monomers nested into one single tree branch have been observed. Interconversion is then achieved by functions that add or delete one monomer independently from the "polymerization degree" of the input functions. Other polymeric forms have their monomers arranged as individual branches emanating from the root. A whole zoo of functions based on polymeric structures has been collected by now. Their detailed description is, nevertheless, beyond the scope of this paper.

These examples illustrate how strongly the organization of interactions in the Turing gas is shaped by the availability of copiers. In the absence of copiers the system still develops connected metabolisms, but based on a rather different "chemistry" that appears to sustain a much higher degree of diversity and innovation.

6.3 TURING GAS WITH PERTURBATIONS

The Turing gas evolves from an initial set of functions through random collisions. Since in the early stages almost every collision is reactive, the dilution flux frequently removes functions that never had an interaction. The system is, therefore, highly sensitive to the particular interaction history. Changing one collision could result in a function that would have otherwise not been produced, thus opening the possibility for a strong sensitivity to initial conditions. In fact, different random number sequences lead very frequently to completely different quasi-stationary states. Simulations have been performed by allowing the gas to increase the number of particles during the initial phase, such that on average every initially present function had the chance to interact. The basic principles of self-organization, however, remain the same. As the accumulation of function chaining leads to closure (section 6.1), or to fluctuating motions around a fixed point (section 6.2), the system's evolution and organization becomes more and more predictable. For example, a primitive operator that has disappeared is lost forever.

A new set of questions arises if the system is exposed to perturbations consisting in the injection of new (random) functions. This can happen in a variety of forms, for example, either constantly as a noisy background, or at specific time intervals. The introduction of a new function creates a center of innovation. New products are generated through secondary reactions spawned by the perturbing agent.

Consider the system without copy reactions discussed in the previous section. The state after 5×10^5 collisions, containing 51 different functions, is taken to be the initial state of a simulation that injects 100 randomly generated functions every 2×10^4 interactions. In order to focus on those functions that organize the system, intersection sets are analyzed that capture the unchanging part during periods lasting between 5×10^4 to 1×10^5 collisions.

The unchanging core between collisions 2×10^5 and 3×10^5 consists of a set $\mathcal{A}$ of 12 functions whose interactions are partly innovative. The interaction graph induced on the vertex set $\mathcal{A} \backslash \Phi[\mathcal{A}]$ is again connected. The structure of the functions is still

polymeric, but the architecture changed with respect to the situation described in section 6.2. The system contains four polymer families, two of which it still shares with the quasi-stationary state described in section 6.2. These groups are induced by the monomers $B1 = (<(*(>*\mathtt{aa}))(+\mathtt{a}))$ and $C1 = (*\,B1)$. The previous polymerizing function $(*(*\mathtt{aa})\mathtt{a})$ has been replaced by $A1 = (*\mathtt{aa})$. $A1$ is the cause of the change in architecture, the polymers still being branched and self-similar, but this time "doublets" of each other. The fourth group is also new and based on the monomer $D1 = (>(*\mathtt{aa}))$.

The new monomers are the products of reactions involving perturbing functions and "resident" ones. Once $(*\mathtt{aa})$ has been produced, its takeover is not a pure accident. In the previous section it was noticed that the polymers induced by the monomer $(*(*\mathtt{aa})\mathtt{a})$ hit the length limitation of 300 characters very soon. The products of an agent with length l and polymerization factor p reach the length limit after $n = \log(300/l)/\log p$ iterated applications. Under the action of the triplicating monomer with $l = 9$ and $p = 3$, the length limit is reached already after three applications. In case of the duplicating monomer $(*\mathtt{aa})$, $l = 5$ and $p = 2$, the limit is reached after six applications. This shows that the polymer populations induced by $(*\mathtt{aa})$, and that can be sustained given the actual limitations of the system, are much more diverse. In addition, the $(*(*\mathtt{aa})\mathtt{a})$-based populations and the $(*\mathtt{aa})$-based populations cooperate on a short time scale: the action of both monomers on each other as well as suitable depolymerization pathways contribute to their mutual maintainance in the system. In the long run, however, the mere combinatorics of the $(*\mathtt{aa})$-subsystem will provide a richer network that has more vertices and more interaction possibilities. In short, under the actual limitations, a system based on duplications sustains more reactive collisions than a system based on triplications, and will eventually displace the triplicating system. Note that the combinatorial argument holds not only for the polymer family consisting of $(*\mathtt{aa})$-monomers, but, as well, for the other polymeric species built up through the action of $(*\mathtt{aa})$. In fact, the number of subsytems (disconnected in the interconversion sense, not in the functional sense) increased from three to four.

As time proceeds and perturbations are introduced every 2×10^4 collisions, time slices of the duration of 5×10^4 collisions are continously monitored for invariant sets. 1.5×10^5 collisions later relative to the previous quasi-stationary state a new invariant set has been found. The set contains 20 functions whose interactions are highly innovative. Their structure is again based on polymer combinatorics, but this time generating a plethora of structures.

Four monomers are present, three of which are identical to the previous metastable state: $A1 = (*\mathtt{aa})$, $B1 = (<(*(>*\mathtt{aa}))(+\mathtt{a}))$ and $D1 = (>(*\mathtt{aa}))$. The new monomer is $C1 = (*('('\mathtt{a})\mathtt{a})$. Note that now two monomers, $A1$ as well as $C1$, have polymerizing activity. $A1$ doubles its input as usual, while $C1$ joins its input to $('\mathtt{a})$, therefore giving rise to polymers of the form $(('\mathtt{a})('\mathtt{a})\ldots('\mathtt{a})E)$, with E being any expression in the system. A polymer of $A1$ usually interacts differently than the monomer. For example, $((*\mathtt{aa})(*\mathtt{aa}))$ performs the function that results from "the doubled input acting on the doubled input," which is quite different than polymerizing by doubling the input. Functions generated by

$C1$ acting on functions containing $C1$, in contrast, always keep the function of the monomer. For example, $F = ((\text{'a})(\text{'a})(*(\text{'}(\text{'a}))\text{a})$ acts on some input E by producing $((\text{a})(\text{'a})E)$. If the expression E is a form that contains $C1$, it will be active by itself in a similar fashion. Note the transformation of one ('a) into an (a) during the action of F: a new "building block," (a), has been introduced. Suitable interactions give rise to all sorts of (a)-('a)-copolymers. In addition, all these products interact with the polymer families arising from $B1$ and $D1$.

Another series of perturbation exeperiments was performed with systems that allow copying. For example, the set of 13 functions from the quasi-stationary state of the system discussed in section 6.1 was perturbed by 10 randomly chosen functions, each introduced in 10 copies. The system was perturbed by such an event every 10^5 collisions. Half a million collisions later the system became homogeneous containing solely the partial copier (and self-replicator) $(>(*\text{aa}))$.

Systems that forbid copying reactions seem to respond in a much subtler way to perturbations. The system discussed in the previous paragraphs, for example, underwent transitions among meta-stable states characterized by a high degree of diversity. If forcing a base level of cooperativity by disallowing copying always results in systems with a higher adaptability than systems that contain self-replicating functions cannot be answered at present. More systematic investigations are needed.

The above examples are meant to illustrate one type of behavior resulting from the adaptation of a Turing gas to external perturbations. The range of responses will depend on the precise form of the noise level: how many functions in what copy numbers at which times. A systematic study of adaptive responses is beyond the scope of this paper, and leads to a series of questions about techniques that are suitable for monitoring "adaptive activity." A forthcoming paper is addressing these issues.[27]

7. SUMMARY AND CONCLUSIONS

The present contribution is based on the hypothesis that the distinguishing feature of adaptive systems consists in a closed loop between objects from a combinatorial variety and the functions they encode. This paper introduces a simple instance of a new class of models aimed at isolating this feature and to study its consequences.

The key idea is to represent such a loop through a formal computational language. The syntactical and semantical levels of a language fit naturally into the object and function poles of that loop. The language used in this paper maps strings of characters into symbolic algorithms that operate on strings. It is closely related to the λ-calculus invented by Alonzo Church in the '30s, and its actual implementation is derived from Gregory Chaitin's permissive toy LISP.[4]

As a first step, the algorithms have been restricted to be functions in one variable. A character string represents a function that acts on a single other character string. The model is complete in the sense that the interaction between functions is

embedded into the language itself. In this paper, interaction is naturally defined to be function composition, and is, therefore, asymmetric. The result of an interaction is a syntactically legal string that encodes a possibly new function.

An iterated map is defined by repeatedly applying the interaction operator to a given set of functions. The concept of an interaction graph is introduced as a representation of the iterated map. The fixed points of an interaction graph are closed, self-maintaining sets. The characterization of all finite self-maintaining sets of functions is an open mathematical problem. A solution to this problem is important with respect to a classification of cooperative structures, and might bear some biological significance (see section 8). Seeding sets are defined as subgraphs that can regenerate the original graph under the action of the map. If the original graph is a fixed point of the map, the seeding sets determine its stability. Some properties of innovative interaction graphs have been discussed in section 4.

The interaction graph, and, equivalently, the iterated map, describe a dynamical system induced by the language on the power set of functions. This graph dynamics is supplemented by a system of mass action kinetics leading to a frequency distribution on the set of functions in a graph. The kinetics is induced through a stochastic process termed "Turing gas." A Turing gas consists of a fixed number of function particles that are randomly chosen for pairwise collisions. In the present scheme, a reactive collision keeps the interaction partners in addition to the reaction product. A stochastic unspecific dilution flux is provided to hold the number of particles in the system at a predefined value.

Three versions of the Turing gas have been discussed. In one version, the time evolution of the gas is observed after its initialization with N randomly generated functions. In the second version, the collision rule is changed to forbid reactions resulting in a copy of one of the collision partners. In the third version, the gas is allowed to settle into a quasi-stationary state, where it is perturbed by injecting new random functions.

The results can be summarized as follows:

1. Ensembles of initially random functions self-organize into ensembles of specific functions sustaining cooperative interactions. Self-replicators, parasites, general copy functions, as well as partial copiers, shape the dynamics of the system. The "innovation rate," i.e., the frequency of collisions that result in functions not present in the system, decreases with time indicating a steady closure with respect to interactions. If the stochastic process is left to itself after injecting the initial functions, it will eventually hit an absorbing barrier characterized by a single replicator type, by a possibly hetergeneous mixture of non-reactive functions ("dead system"), or by a self-reproducing set where each individual function species is a seeding set. The system typically exhibits extremely long transients characterized by mutually stabilizing interaction patterns. Such patterns include a hierarchical organization of interacting self-maintaining sets. Sometimes these subsets are disconnected from each other with respect to interconversion pathways, but connected with respect to functional couplings.

2. Forbidding copy reactions results in a new type of cooperative organization as compared to the case in which copy reactions and, therefore, self-replicators were allowed. The system switches to functions based on a "polymeric" architecture that entertain a closed web of mutual synthesis reactions. The individual functions are usually organized into disjoint subsets of polymer families based on distinct monomers. As in the case of copy reactions, these subsets interact along specific pathways leading to a cooperativity at the set level. Due to the polymeric structure of the functions, the Turing gas remains innovative. A much higher degree of diversity and stability is achieved than in systems that are dominated by self-replicators.
3. An open system is modeled by introducing new random functions that perturb a well-established ecology. In the case without copy reactions, the system underwent transitions among several new quasi-stationary states, each characterized by an access to higher diversity. Systems with copy reactions were more vulnerable to perturbations and lost in the long run much of their structure.

The main conclusions are:

1. A formal computational language captures basic qualitative features of complex adaptive systems. It does this because of
   a. a powerful, abstract and consistent description of a system at the "functional" level, due to an unambiguous mathematical notion of function;
   b. a finite description of an infinite (countable) set of functions, therefore providing a potential for functional open-endedness; and
   c. a natural way of enabling the construction of new functions through a consistent definition of interaction between functions.
2. Populations of individuals that are both an object at the syntactic level and a function at the semantic level, give rise to the spontaneous emergence of complex, stable, and adaptive interactions among their members.

The present contribution raises many questions. The next section lists some of these questions, briefly points to related work, and considers future directions.

8. OUTLOOK

One feature of the present model is the coupling of a dynamics governing the topology of an interaction graph with a dynamics governing a frequency distribution over its vertices. The present model is by no means the first to exhibit such a structure, although it differs fundamentally in approach and scope from others.

Approaches based on this coupling have been proposed in several areas of research. D. Farmer, S. Kauffman, and N. Packard,[11] as well as R. Bagley et al.[1] considered a model of polymers intended to be RNA strands or proteins, undergoing

condensation and hydrolysis reactions catalyzed by other polymers. S. Rasmussen et al.[26,28] investigated a model involving random catalytic connections of the hypercycle type,[10] R. deBoer and A. Perelson[7] proposed a model of the immune system that exhibits the above coupling, and J. Holland's classifier system[16] is an instance based on a genetic scheme.

More abstract approaches are considered by J. McCaskill[24] and S. Rasmussen et al.[29] Rasmussen's system consists of generalized assembler code instructions that interact in parallel inside a controlled computer memory giving rise to cooperative phenomena. McCaskill uses binary strings to encode transition-table machines of the Turing type that read and modify bit strings. The *ansatz* developed in this paper emerged from the direct experience with McCaskill's system, from several attempts to redefine it, and from the observation that Rasmussen's system shares too many properties with cellular automata.

Cellular automata provide a powerful approach to the study of the emergence of loops between objects and functions. Incidentally, John von Neumann envisioned such a loop when invoking symbolic instruction and universal construction as necessary conditions for the evolution of complexity.[34] Much work has been done to understand the conditions that allow for its emergence.[14,20,21,35] It becomes, however, difficult to study the consequences of such a loop at the same level of description that has been used to study its emergence.

Some issues that accumulated throughout the paper are addressed in the following. One concerns the asymmetry of the interactions. The immediate observation is that many complex systems in Nature have asymmetric interactions. Neurons, other cell types, and species are but a few examples. Clearly, a strict analogy with molecules is not possible within the present collision rule. The decision of which molecule, in a given pair, is the enzyme and which molecule is the substrate cannot be reversed in later collisions. This could be taken into account by redefining the collision rule such as to compute both expressions, $f(g)$ as well as $g(f)$, and taking, for example, the longest expression to be the collision product (ties resolved lexicographically).

Enzyme and substrate interpretations become stretched, as soon as general interaction schemes, $\Phi(f,g)$, are considered. An open question is if interactions Φ, different from chaining, result in qualitatively similar patterns of organization.

One of the most obvious generalizations of the system is the multivariable case. Considering functions with up to n variables is interesting for at least three reasons. First, it leads naturally to $(n+1)$-body interactions. Second, the system can self-organize with respect to the density of 1-, 2-, ..., n-variable functions. Third, suppose the interaction is still given by a generalized function composition. A two-variable function $f(x,y)$, then interacts with functions g and h (in this order) by producing $i = f(g,h)$. f acts, with respect to any pair g and h, precisely like a binary interaction law expression Φ did previously (see section 3.3). However, f can now be modified through interactions with other components of the same system. This might have consequences for the architecture of organizational patterns that are likely to evolve. The extension to n variables is currently in preparation.

Further questions relate to the number and type of primitive operators. An extended set of experiments was conducted with 11 operators (including the six reported here). The additional primitives involved instructions like calls to the interpreter from within the interpretation process. The basic results are very similar with respect to organizational patterns. Nevertheless, more powerful function constructors should be taken into account depending on the particular application.

The dependence of the observed phenomena on the number of particles in the system has not been considered so far. Do new phenomena appear if the system consists of a million particles?

Spatial systems have not been considered here. Nor have genetic mechanisms. What happens if noise is introduced through error-prone execution with some per step error rate q? What concepts of "mutation" can be envisioned?

Slight variations in the semantics of the language had no effect on the basic results. Nothing can be said at present about language architectures other than the λ-calculus. This raises an important question: Are the phenomena reported in this paper universal with respect to representation? Do different representations only affect the probability by which particular organizational patterns emerge?

Rate constants were not considered in this contribution. The a_{kij} in Eq. (2) were unity if an interaction between i and j resulted in k, and zero otherwise. The model puts the emphasis on relationships among agents rather than on kinetic details. Nevertheless, "rate constants" can be added, e.g., by considering the space-time resources required to complete the computation of an interaction.

Functional interactions based on chemistry enabled the evolution of complex adaptive systems of the biological type. Biological systems seem to have various levels at which "higher-order" structures interact again in a functional way. For example, cell types and control mechanisms give rise to ontogeny in multicellular organisms. The results reported in this contribution suggest that a formal language might be a useful model for systems that are characterized by functional interactions.

The new model class introduced in this paper raises sensible mathematical questions that are interesting by and of themselves. However, the point of view outlined here is useful only if it can organize knowledge about the real world. For example, by being predictive or by enabling a logical characterization or classification of certain phenomena that would otherwise remain a pure matter of fact. The viewpoint would be useful, for instance, if a characterization of finite self-maintaining sets of functions in up to n variables exhibits a subset with basic similarities to known biological life-cycle organizations. It would be useful, for instance, if such a system (with suitable modifications) allows the study and the understanding of the entangled interplay between rules of selection and products of selection.

Doyne Farmer recently drew my attention to a paper by O. E. Rössler[31] that appeared in German in 1971. In that remarkable paper Rössler expands precisely on the chemical metaphor I had in mind when developing the work presented in this contribution. Although Rössler did neither present a specific model nor did he invoke the constructive aspect of computable functions, his paper clearly states some of the main thoughts that motivated the present work.

ACKNOWLEDGMENTS

This work would have hardly been possible without inspiring discussions with John McCaskill, Wojciech Zurek, David Cai, Norman Packard, Steen Rasmussen, David Lane, Doyne Farmer, Stuart Kauffman, Leo Buss, David Sharp, Gian-Carlo Rota, Peter Stadler, Peter Schuster, Jeff Davitz, and Chris Langton.

This work has been performed under the auspices of the United States Department of Energy.

A. ALCHEMY IS A VARIANT OF PURE LISP

A.1 SYNTAX

Let $\mathcal{C}$ be an alphabet that includes left and right parentheses. All characters except left and right parentheses are called atoms. The fundamental syntactic structure is termed "expression."

DEFINITION A.1 An expression is an atom or a list.

The structure of a list is defined recursively.

DEFINITION A.2 A list consists of a left parenthesis followed by zero or more atoms, or lists followed by a right parenthesis.

Let $\mathcal{C} = \{\texttt{a,b,c,d,e,(,)}\}$, then `()`, `(c)`, `(((dba)(b))b(ec))` are lists.

Lists can be represented by ordered trees with the leaves being the atoms (Figure 1). A matching pair of left and right parentheses represents an internal node. The empty list is treated like an atom, but consists of two characters. A list is reconstructed from a tree by traversing the tree in preorder, i.e., by visiting the root of the first (leftmost) tree, traversing the subtrees of the first tree in preorder, and traversing the remaining trees in preorder.

Expressions are self-delimiting. An expression is complete as soon as the number of right parentheses matches the number of left parentheses. For instance, the string `(ab)c(de)` is not an expression. The point is that the parentheses do not balance for the first time at the end of the character string, but at the end of `(ab)`.

A.2 SEMANTICS

A.2.1 VALUE The basic semantical concept is that of the "value" V of an expression. The evaluation of an expression refers always to an "association list" L. An association list is a look-up table that stores zero or more pairs. The first element of such a pair is always an atom, and the second element is an expression that is to be substituted for that atom during the evaluation process. If a given atom does not appear in the first position of any pair, then this atom evaluates to itself. In an empty association list, $L =$`()`, every atom evaluates to itself. The empty list is considered to be an atom. Its value is always the empty list.

Value assignments in the association list are indicated by a left arrow. For instance, the pair that assigns to `a` the value `b` is written as `a`$\leftarrow$`b`. With the association list

$$L = (\texttt{d} \leftarrow (\texttt{cd(a)})\ \texttt{e} \leftarrow \texttt{b}\,\texttt{a} \leftarrow (\texttt{e}))\ ,$$

the expression `e` evaluates to `b`. For this I shall write: $\mathbf{V}[\texttt{e}, L] =$`b`. Furthermore, $\mathbf{V}[\texttt{d}, L] =$`(cd(a))`, $\mathbf{V}[\texttt{a}, L] =$`(e)`, and $\mathbf{V}[\texttt{b}, L] =$`b`.

DEFINITION A.3 The value of an atom is obtained from the association list L. The value of a non-atomic expression is obtained recursively in terms of the values of its elements (atoms and/or lists).

A list is, therefore, parsed into its constituent expressions, and each expression is evaluated. The concept of "function" is needed to further understand the evaluation process.

A.2.2 FUNCTIONS AND EVALUATION Two types of functions are distinguished. Primitive and defined (or composite) functions.

Primitive functions are the predefined operators of the language. Their names are atoms from the character set $\mathcal{C}$ reserved for this purpose.

DEFINITION A.4 An n-ary primitive operates on n arguments that are elements of the same list to which the operator belongs. The n arguments are obtained by evaluating the n expressions following the atom denoting the primitive.

If the list contains less than n expressions, an empty list is supplied for each missing one. Any expressions in excess are ignored. The primitive operators are defined in Appendix B. Every operator returns an expression.

A defined function is the value of an expressions. This value is applied to arguments. The arguments are obtained in exactly the same way as in the case of the primitives.

DEFINITION A.5 In AlChemy the values of all expressions that are not arguments to primitive operators define functions.

The exception ensures that a primitive operator performs its manipulations, without side effects arising from interacting arguments.

In order for a function to operate on arguments, the variables have to be specified.

DEFINITION A.6 In AlChemy all atoms not denoting operators are variables.

DEFINITION A.7 "Applying a defined function to arguments" means to evaluate the function expression after having substituted every occurrence of the variables with the corresponding arguments.

Technically this is a two-step process: (1) Updating the association list by appending the old list to a new list of pairs that binds the variables with the corresponding argument expressions, and (2) evaluating the function expression thereby using the new association list.

In this paper only the simplest case is considered: functions in one variable; hence, there is only one more atom in addition to the primitive operators. Throughout this paper the atom a denotes the variable. The generalization to multivariable functions is straightforward.

The following is a formal definition of the value of an expression. The association list L contains, at any time, the current value of the variable, and is of the form

L =(a← Z), where Z is some expression. The empty association list L =() is equivalent to L =(a←a). Consider the list $(f_1\ f_2\ \ldots\ f_n)$ with an empty association list. According to definition A.5 the values of the expressions $f_1, f_2, \ldots, f_n$ define functions.

DEFINITION A.8 A list consisting of more than one function definition is evaluated by applying each function to its arguments in turn. The value of the list is the expression obtained by appending the result of each function application to an initially empty list. Results consisting of an empty list are ignored.

According to this definition the value of the list, $(f_1\ f_2\ \ldots\ f_n)$ is given by

$$\mathbf{V}[(f_1\ f_2\ \ldots\ f_n), L] = \begin{cases} \text{application of k-ary primitive,} & \text{if } f_1 \in \mathcal{O}; \\ (\mathbf{V}[\mathbf{V}[f_1, L], (\mathtt{a} \leftarrow \mathbf{V}[f_2, L])] \\ \ \ \mathbf{V}[\mathbf{V}[f_2, L], (\mathtt{a} \leftarrow \mathbf{V}[f_3, L])] \\ \ \ \vdots \\ \ \ \mathbf{V}[\mathbf{V}[f_{n-1}, L], (\mathtt{a} \leftarrow \mathbf{V}[f_n, L])]), & \text{otherwise,} \end{cases}$$

where $\mathcal{O}$ denotes the set of primitive operators. Usually, L = (). Redundant parentheses are always removed, e.g., $((f))$ becomes (f). In addition, if an evaluation results in a function body consisting of the parenthesized variable, $\mathbf{V}[f_i, L] = (\mathtt{a})$, the parentheses are removed. The application of (a) would mostly result in (), since an empty list is supplied for the missing argument.

This procedure contains, in contrast to pure LISP, a "sequential" aspect. A purely recursive scheme would evaluate an expression $f = (ghi)$ by first applying the value of h to the value of i, and then applying the value of g to that result. The departure from a purely recursive scheme has been decided by observing that the nesting of too many evaluations considerably shortens the output expressions, often leading to empty lists. The sequential mode is an efficient way to offset this effect without constraining the combinatorics of expressions.

Stated in terms of tree structures, evaluating an expression means evaluating its tree. The value of any node is obtained by applying the value of each subtree to its right-neighbor sibling in turn, each time appending the result to the current value expression. If the leftmost subtree of a node is an n-ary operator, then the value of that node is simply obtained by applying the primitive operation to the values of the n siblings of that operator. The value of a tree is therefore the value at its root.

A.2.3 INTERACTIONS BETWEEN ALCHEMY FUNCTIONS The interaction between two expressions, f and g, is defined by an expression that has as value the expression $f(g)$. Definition A.5 states that a function is represented by the value of an expression, not the expression itself. To build an expression with value $f(g)$ the quote-operator, $'$, is used. The action of the quote-operator is to prevent evaluation of its argument. Hence, $\mathbf{V}[('f), L] = f$, for all L. This leads to the following interaction expression:

DEFINITION A.9 The interaction between two expressions, f and g, is defined by the expression

$$((' f)(' g))$$

The result of an interaction is given by

$$\mathbf{V}[(('f)('g)), ()] = (\mathbf{V}[f, (\mathtt{a} \leftarrow g)]) .$$

The functions of the model described in this paper have at most one variable; hence, only binary interactions have to be considered. A function of n variables can interact with n other functions.

B. PRIMITIVE OPERATORS

AlChemy's character set is given by

$$\mathcal{C} = \{(\,,)\,,\mathtt{a}\,,+,-,>,<,*,'\}.$$

AlChemy has six primitive operators. These operators manipulate list structures. The operators are not "orthogonal," in the sense that some of them can be expressed through a combination of others.

$+$ Name : head Arguments : 1

Application of this operator to a list returns the first expression of that list. Application to an atom returns the atom itself. The head of an empty list is, therefore, an empty list.

$-$ Name : tail Arguments : 1

Application of this operator to a list returns what remains after the first expression of that list is deleted. Application to an atom returns the atom itself. The tail of an empty list is, therefore, an empty list.

$>$ Name : inversehead Arguments : 1

Application of this operator to a list returns the last expression of that list. Application to an atom returns the atom itself. The inverse head of an empty list is, therefore, an empty list.

$<$ Name : inversetail Arguments : 1

Application of this operator to a list returns what remains after the last expression of that list is deleted. Application to an atom returns the atom itself. The inverse tail of an empty list is, therefore, an empty list.

$+$ Name : join Arguments : 2

If the second argument is an atom, but not the empty list, then the operator returns the first argument. If the second argument is an n-element list, then the result is an $n+1$-element list whose head is the first argument and whose tail is the second argument.

Exception: If the second argument is an expression E of the form $E = (\text{"op"}\ldots)$, where "op" is one of the primitive operators defined in this section, then E is wrapped into parentheses, (E), prior to application of the join operator.

Remark: Let, for example, (A) and $(+B)$ be expressions that are to be joined. The exception rule applies to $(+B)$. The result will then simply be $((A)(+B))$. If this product is to be further evaluated, then the group (A) will be a function acting on the group $(+B)$. Without the exception rule the join product would have been $((A)+B)$. Further evaluation would have (A) act on the character $+$, which is less interesting. Applying the exception rule means confining the system to a subset of the computations that would occur otherwise.

Name : quote Arguments : 1

The operator returns the unevaluated argument expression.

C. AN EVALUATION EXAMPLE

Consider the interaction expression E represented by the middle tree in Figure 3.

$$E = (A\ B) \tag{C.1}$$

with

$$A = ('((+\mathtt{a})(*(>\mathtt{a})(>(+\mathtt{a}))))),$$
$$B = ('((*(+\mathtt{a})(*(-\mathtt{a})(<\mathtt{a})))\mathtt{a})).$$

The value of Eq. (C.1) is

$$\mathbf{V}[E, \mathtt{a} \leftarrow \mathtt{a}] = (\mathbf{V}[\mathbf{V}[A, \mathtt{a} \leftarrow \mathtt{a}], \mathtt{a} \leftarrow \mathbf{V}[B, \mathtt{a} \leftarrow \mathtt{a}]]). \tag{C.2}$$

$\mathbf{V}[A, \mathtt{a} \leftarrow \mathtt{a}]$, with $A = ('F)$ is obtained by applying the quote operator, $'$, to its argument F. Clearly,

$$\mathbf{V}[('F), \mathtt{a} \leftarrow \mathtt{a}] = F = ((+\mathtt{a})(*(>\mathtt{a})(>(+\mathtt{a})))).$$

The same holds for $B = ('G)$:

$$\mathbf{V}[('G), \mathtt{a} \leftarrow \mathtt{a}] = G = ((*(+\mathtt{a})(*(-\mathtt{a})(<\mathtt{a})))\mathtt{a}).$$

Therefore, Eq. (C.2) becomes

$$\mathbf{V}[E, \mathtt{a} \leftarrow \mathtt{a}] = (\mathbf{V}[F, \mathtt{a} \leftarrow G]). \tag{C.3}$$

Write

$$F = (F_1 F_2),$$

with

$$F_1 = (+\mathtt{a})$$
$$F_2 = (*(>\mathtt{a})(>(+\mathtt{a}))),$$

and

$$G = (G_1 G_2),$$

with

$$G_1 = (*(+\mathtt{a})(*-\mathtt{a})(<\mathtt{a}))$$
$$G_2 = \mathtt{a}.$$

Then Eq. (C.2) becomes

$$\mathbf{V}[E, \mathtt{a} \leftarrow \mathtt{a}] = \mathbf{V}[F, \mathtt{a} \leftarrow G] = \mathbf{V}[(F_1 F_2), \mathtt{a} \leftarrow (G_1 G_2)] =$$
$$= (\mathbf{V}[\mathbf{V}[F_1, \mathtt{a} \leftarrow (G_1 G_2)], \mathtt{a} \leftarrow \mathbf{V}[F_2, \mathtt{a} \leftarrow (G_1 G_2)]]). \tag{C.4}$$

$\mathbf{V}[F_1, \mathtt{a} \leftarrow (G_1 G_2)]$ with $F_1 = (+\mathtt{a})$ is obtained by substituting $(G_1 G_2)$ for $\mathtt{a}$ and by applying the "+" operator (see Appendix B):

$$\mathbf{V}[(+\mathtt{a}), \mathtt{a} \leftarrow (G_1 G_2)] = G_1 . \tag{C.5}$$

$\mathbf{V}[F_2, \mathtt{a} \leftarrow (G_1 G_2)]$ is obtained by applying the "$*$" operator to $\mathbf{V}[(>\mathtt{a}), \mathtt{a} \leftarrow (G_1 G_2)]$ and $\mathbf{V}[(>(+\mathtt{a})), \mathtt{a} \leftarrow (G_1 G_2)]$, the values of its arguments $(>\mathtt{a})$ and $(>(+\mathtt{a}))$, respectively. Refer to appendix B for the action of the primitive operator ">."

$$\mathbf{V}[(>\mathtt{a}), \mathtt{a} \leftarrow (G_1 G_2)] = (G_2) = (\mathtt{a}) \tag{C.6};$$
$$\mathbf{V}[(>(+\mathtt{a})), \mathtt{a} \leftarrow (G_1 G_2)] = (*(-\mathtt{a})(<\mathtt{a})) . \tag{C.7}$$

Hence,

$$\mathbf{V}[F_2, \mathtt{a} \leftarrow (G_1 G_2)] = ((\mathtt{a})(*(-\mathtt{a})(<\mathtt{a}))) . \tag{C.8}$$

Equation (C.2) is now

$$\begin{aligned} \mathbf{V}[E, \mathtt{a} \leftarrow \mathtt{a}] = & \mathbf{V}[F, \mathtt{a} \leftarrow G] = \\ & = (\mathbf{V}[\mathbf{V}[F_1, \mathtt{a} \leftarrow (G_1 G_2)], \mathtt{a} \leftarrow \mathbf{V}[F_2, \mathtt{a} \leftarrow (G_1 G_2)]]) = \\ & = (\mathbf{V}[(*(+\mathtt{a})(*(-\mathtt{a})(*\mathtt{a}))), \mathtt{a} \leftarrow ((\mathtt{a})(*(-\mathtt{a})(<\mathtt{a}))) \end{aligned} \tag{C.9}$$

which means applying "$*$" to

$$\mathbf{V}[(+\mathtt{a}), \mathtt{a} \leftarrow ((\mathtt{a})(*(-\mathtt{a})(<\mathtt{a})))] = (\mathtt{a}), \tag{C.10}$$

and

$$\mathbf{V}[(*(-\mathtt{a})(<\mathtt{a})), \mathtt{a} \leftarrow ((\mathtt{a})(*(-\mathtt{a})(<\mathtt{a})))] = ((*(-\mathtt{a})(<\mathtt{a}))\mathtt{a}) \tag{C.11}$$

resulting from the application of "$*$" to $(*(-\mathtt{a})(<\mathtt{a}))$, the value of $(-\mathtt{a})$ in Eq. (C.11), and to $(\mathtt{a})$, the value of $(<\mathtt{a})$ in (11). The application of "$*$" to Eqs. (C.10) and (C.11) finally completes

$$\mathbf{V}[E, \mathtt{a} \leftarrow \mathtt{a}] = \mathbf{V}[F, \mathtt{a} \leftarrow G] = ((\mathtt{a})(*(-\mathtt{a})(<\mathtt{a}))\mathtt{a}) . \tag{C.12}$$

Equation (C.12) is the collision product of functions F and G as shown in Figure 3.

REFERENCES

1. Bagley, R. J., J. D. Farmer, S. A. Kauffman, N. H. Packard, A. S. Perelson, I. M. Stadnyk. "Modeling Adaptive Biological Systems." *BioSystems* **23** (1989): 113–138.
2. Barendregt, H. P. *The Lambda Calculus*, Studies in Logic and the Foundations of Mathematics, vol. 103. Amsterdam: North Holland, 1984.
3. Buss, L. W. "The Evolution of Individuality." Princeton: Princeton University Press, (1987): 196.
4. Chaitin, G. J. *Algorithmic Information Theory.* Cambridge: Cambridge University Press, 1987.
5. Church, A. "An Unsolvable Problem of Elementary Number Theory." *Am. J. Math.* **58** (1936): 345–363.
6. Church, A. "The Calculi of Lambda-Conversion." *Annals of Mathematics Studies*, no. 6. Princeton: Princeton University Press, 1941.
7. deBoer, R., and A. S. Perelson. "Size and Connectivity as Emergent Properties of a Developing Immune Network." *J. Theor. Biol.* (1990): in press.
8. Delbrück, M. *Trans. Conn. Acad. Arts Sci.* **38** (1949): 173–190.
9. Eigen, M. "Self-Organization of Matter and the Evolution of Biological Macromolecules." *Naturwissenschaften* **58** (1971): 465–523.
10. Eigen, M., and P. Schuster. *The Hypercycle.* Berlin: Springer, 1979.
11. Farmer, J. D., S. A. Kauffman, and N. H. Packard. "Autocatalytic Replication of Polymers." *Physica D* **22** (1986): 50–67.
12. Fontana, W. "Algorithmic Chemistry: A New Approach to Functional Self-Organization." Technical Report LA-UR 90-3431, Los Alamos National Laboratory. Submitted to *Physica D*, 1990.
13. Gödel, K. Presented in his 1934 lectures at the Institute for Advanced Study. Quoted from: S. Kleene. "Turing's Analysis of Computabity and Major Applications of it." In *The Universal Turing Machine: A Half-Century Survey*, edited by R. Herken, 17–54. Oxford: Oxford University Press, 1988.
14. Gutowitz, H. A., B. K. Knight, and J. D. Victor. *Physica D* **48** (1987): 18.
15. Hofbauer, J., and K. Sigmund. *The Theory of Evolution and Dynamical Systems.* Cambridge: Cambridge University Press, 1988.
16. Holland, J. H. "Escaping Brittleness: The Possibilities of General Purpose Learning Algorithms Applied to Parallel Rule-Based Systems." In *Machine Learning II*, edited by R. S. Mishalski, J. G. Carbonell, and T. M. Mitchell, 593–623. Kaufman, 1986.
17. Kauffman, S. A. "Autocatalytic Sets of Proteins." *J. Theor. Biol.* **119** (1986): 1–24.
18. Kauffman, S. A. *J. Cybernetics* **1** (1971): 71–96.
19. Lane, D., J. H. Holland, and W. Fontana. Working paper under the auspices of the Adaptive Computation Program. Santa Fe, 1990.
20. Langton, C. G. *Proceedings of the 1989 Cellular Automata Workshop*, edited by H. A. Gutowitz. North-Holland, 1990.

21. Li, W., and N. H. Packard. *Complex Systems* **4** (1990): 281–297.
22. Manes, E. G., and M. A. Arbib. *Algebraic Approaches to Program Semantics.* Berlin: Springer-Verlag, 1986.
23. Mayr, E. *The Growth of Biological Thought.* Cambridge, MA: Harvard University Press, 1982.
24. McCaskill, J. S. In preparation.
25. Miller, J. H., W. Fontana, and P. Schuster. "Towards a Mathematics of a Turing Gas." In preparation, 1990.
26. Mosekilde, E., S. Rasmussen, and T. S. Sorensen. "Self-Organization and Stochastic Re-Causalization in Systems Dynamics Models." In *Proceedings of the 1983 International Systems Dynamics Conference*, 1983, 123.
27. Packard, N. H., and W. Fontana. In preparation, 1990.
28. Rasmussen, S. "Toward a Quantitative Theory of the Origin of Life." In *Artificial Life*, edited by C. G. Langton. Santa Fe Institute Studies in the Sciences of Complexity, Vol. VI, 79–104. Reading, MA: Addison-Wesley, 1988.
29. Rasmussen, S., C. Knudsen, R. Feldberg, and M. Hindsholm. "The Coreworld: Emergence and Evolution of Cooperative Structures in a Computational Chemistry." *Physica D* **42** (1990): 111–134.
30. Reingold, E. M., J. Nievergelt, and N. Deo. *Combinatorial Algorithms: Theory and Practice.* Englewood Cliffs, NJ: Prentice-Hall, 1977.
31. Rössler, O. E. "A System Theoretic Model of Biogenesis." *Zeitschrift für Naturforschung* **26b** (1971): 741–746.
32. Stadler, P. F., and P. Schuster "Mutation in Autocatalytic Reaction Networks." Working paper number 90-022, Santa Fe Institute. Submitted to *J. Math. Biol.*, 1990.
33. Turing, A. M. "On Computable Numbers with an Application to the Entscheidungs Problem." *P. Lond. Math. Soc. (2)* **42** (1936-1937): 230–265.
34. von Neumann, J. *Theory of Self-Reproducing Automata*, edited by A. W. Burks. Urbana: University of Illinois Press, 1966.
35. Wolfram, S. *Physica D* **10** (1984): 1.