

Life as Evolving Software*

Gregory Chaitin[†]

January 1, 2012

1 Turing as a Biologist

Few people remember Turing's work on pattern formation in biology (morphogenesis), but Turing's famous 1936 paper "On Computable Numbers. . ." exerted an immense influence on the birth of molecular biology indirectly, through the work of John von Neumann on self-reproducing automata, which influenced Sydney Brenner who in turn influenced Francis Crick, the Crick of Watson and Crick, the discoverers of the molecular structure of DNA. Furthermore, von Neumann's application of Turing's ideas to biology is beautifully supported by recent work on evo-devo (evolutionary developmental biology). The crucial idea: DNA is multi-billion year old software, but we could not recognize it as such before Turing's 1936 paper, which according to von Neumann creates the idea of computer hardware and software.

We are attempting to take these ideas and develop them into an abstract fundamental mathematical theory of evolution, one that emphasizes biological creativity, inventiveness and the generation of novelty. This work is being published in two parts.

Firstly a non-technical book-length treatment: G. Chaitin, *Proving Darwin: Making Biology Mathematical* to be published by Pantheon in 2012. There we explain at length the basic concepts and the history of ideas. For an overview of this book, a lecture entitled "Life as evolving software," go to www.youtube.com and search for *chaitin ufrgs*.

And in this paper we present a technical discussion of the mathematics of this new way of thinking about biology. More precisely, we present an information-theoretic analysis of Darwin's theory of evolution, modeled as a hill-climbing algorithm on a fitness landscape. Our space of possible organisms consists of computer programs, which are subjected to random mutations. We study the random walk of increasing fitness made by a single mutating organism. In two

*Intended as a contribution to the Turing Centenary volume H. Zenil, *A Computable Universe*, World Scientific, to appear.

[†]Chaitin is a CAPES visiting professor at the Federal University of Rio de Janeiro, an honorary professor at the University of Buenos Aires and a member of the Académie Internationale de Philosophie des Sciences. His email address is gjchaitin@gmail.com.

different models we are able to show that evolution will occur and to characterize the rate of evolutionary progress, i.e., the rate of biological creativity.

We call this new theory *metabiology*, and it deals with the evolution of mutating software and with random walks in software space. The mathematics we use is essentially Turing's version of computability theory from the 1930s, including his colorful oracles, plus the idea of how to associate probabilities with computer programs utilized since the 1970s in algorithmic information theory, which is summarized in the appendix of this paper.

It remains to be seen how far these ideas will go, but as is shown in this paper and in the companion volume [13], the first steps are encouraging. In our opinion, Turing's ideas are of absolutely fundamental importance in biology, since biology is all about digital software.

2 Introduction to this Paper

For many years we have been disturbed by the fact that there is no fundamental mathematical theory inspired by Darwin's theory of evolution [1, 2, 3, 4, 5, 6, 7, 8, 9]. This is the fourth paper in a series [10, 11, 12] attempting to create such a theory.

In a previous paper [10] we did not yet have a workable mathematical framework: We were able to prove two not very impressive theorems, and then the way forward was blocked. Now we have what appears to be a good mathematical framework, and have been able to prove a number of theorems. Things are starting to work, things are starting to get interesting, and there are many technical questions, many open problems, to work on.

So this is a working paper, a progress report, intended to promote interest in the field and get others to participate in the research. There is much to be done.

In order to present the ideas as clearly as possible and not get bogged down in technical details, the material is presented more like a physics paper than a math paper. Estimates are at times rather sloppy. We are trying to get an idea of what is going on. The arguments concerning the basic math framework are however very precise; that part is done more or less like a math paper.

3 History of Metabiology

In the first paper in this series [10] we proposed modeling biological evolution by studying the evolution of randomly mutating software—we call this *metabiology*. In particular, we proposed considering a single mutating software organism following a random walk in software space of increasing fitness. Besides that the main contribution of [10] was to use the Busy Beaver problem to challenge organisms into evolving. The larger the positive integer that a program names, the fitter the program.

And we measured the rate of evolutionary progress using the Busy Beaver function $BB(N)$ = the largest integer that can be named by an N -bit program. Our two results employing the framework in [10] are that

- with random mutations, random point mutations, we will get to fitness $BB(N)$ in time exponential in N (evolution by *exhaustive search*) [10, 11],
- whereas by choosing the mutations by hand and applying them in the right order, we will get to fitness $BB(N)$ in time linear in N (evolution by *intelligent design*) [11, 12].

We were unable to show that *cumulative evolution* will occur at random; exhaustive search starts from scratch each time.¹

This paper advances beyond the previous work on metabiology [10, 11, 12] by proposing a better concept of mutation. Instead of changing, deleting or inserting one or more adjacent bits in a binary program, we now have high-level mutations: we can use an arbitrary algorithm M to map the organism A into the mutated organism $A' = M(A)$. Furthermore, the probability of the mutation M is now furnished by algorithmic information theory: it depends on the size in bits of the self-delimiting program for M . It is very important that we now have a natural, universal probability distribution on the space of all possible mutations, and that this is such a rich space.

Using this new notion of mutation, these much more powerful mutations, enables us to accomplish the following:

- We are now able to show that *random evolution will become **cumulative*** and will reach fitness $BB(N)$ in time that grows roughly as N^2 , so that random evolution behaves much more like intelligent design than it does like exhaustive search.²
- We also have a version of our model in which we can show that ***hierarchical structure will evolve***, a conspicuous feature of biological organisms that previously [10] was beyond our reach.

This is encouraging progress, and suggests that we may now have the correct version of these biology-inspired concepts. However there are many serious lacunae in the theory as it currently stands. It does not yet deserve to be called a *mathematical theory of evolution and biological creativity*; at best, it is a sketch of a possible direction in which such a theory might go.

On the other hand, the new results are encouraging, and we feel it would be inappropriate to sit on these results until all the lacunae are filled. After all,

¹The Busy Beaver function $BB(N)$ grows faster than any computable function. That evolution is able to “compute” the uncomputable function $BB(N)$ is evidence of creativity *that cannot be achieved mechanically*. This is possible only because our model of evolution/creativity utilizes an uncomputable Turing oracle. Our model utilizes the oracle in a highly constrained manner; otherwise it would be easy to calculate $BB(N)$.

²Most unfortunately, it is not yet demonstrated that random evolution cannot be as fast as intelligent design.

that would take an entire book, since metabiology is, or will hopefully become, a rich and entirely new field.

That said, the reader will understand that this is a working paper, a progress report, to show the direction in which the theory is developing, and to indicate problems that need to be solved in order to advance, in order to take the next step. We hope that this paper will encourage others to participate in developing metabiology and exploring its potential.

4 Modeling Evolution

4.1 Software Organisms

In this paper we follow a metabiological [10, 11, 12, 13] approach: Instead of studying the evolution of actual biological organisms we study the evolution of software subjected to random mutations. In order to do this we use tools from algorithmic information theory (AIT) [14, 15, 16, 17, 18, 19]; **to fully understand this paper expert understanding of AIT is unfortunately necessary** (see the outline in the Appendix).

As our programming formalism we employ one of the optimal self-delimiting binary universal Turing machines U of AIT [14], and also, but only in Section 8, a primitive FORTRAN-like language that is not universal.

So our organisms consist on the one hand of arbitrary self-delimiting binary programs p for U , or on the other hand of certain FORTRAN-like computer programs. These are the respective software spaces in which we shall be working, and in which we will study hill-climbing random walks.

4.2 The Hill-Climbing Algorithm

In our models of evolution, we define a hill-climbing random walk as follows: We start with a single software organism A and subject it to random mutations until a fitter organism A' is obtained, then subject that organism to random mutations until an even fitter organism A'' is obtained, etc. In one of our models, organisms calculate natural numbers, and the bigger the number, the fitter the organism. In the other, organisms calculate functions that map a natural number into another natural number, and the faster the function grows, the fitter the organism.

In this connection, here is a useful piece of terminology: A mutation M *succeeds* if $A' = M(A)$ is fitter than A ; otherwise M is said to *fail*.

4.3 Fitness

In order to get our software organisms to evolve it is important to present them with a challenge, to give them something difficult to do. Three well-known problems requiring unlimited amounts of mathematical creativity are:

- **Model A:** Naming large natural numbers (non-negative integers) [20, 21, 22, 23],
- **Model B:** Defining extremely fast-growing functions [24, 25, 26],
- **Model C:** Naming large constructive Cantor ordinal numbers [26, 27].

So a software organism will be judged to be more fit if it calculates a larger integer (our Model A, Sections 5, 6, 7), or if it calculates a faster-growing function (our Model B, Section 8). Naming large Cantor ordinals (Model C) is left for future work, but is briefly discussed in Section 9.

4.4 What is a Mutation?

Another central issue is the concept of a mutation. Biological systems are subjected to point mutations, localized changes in DNA, as well as to high level mutations such as copying an entire gene and then introducing changes in it. Initially [10] we considered mutating programs by changing, deleting or adding one or more adjacent bits in a binary program, and postponed working with high-level source language mutations.

Here we employ an extremely general notion of mutation: A mutation is an arbitrary algorithm that transforms, that maps the original organism into the mutated organism. It takes as input the organism, and produces as output the mutated organism. And if the mutation is an n -bit program, then it has probability 2^{-n} . In order to have the total probability of mutations be ≤ 1 we use the self-delimiting programs of AIT [14].³

4.5 Mutation Distance

A second crucial concept is mutation distance, how difficult it is to get from organism A to organism B . We measure this distance in bits and it is defined to be $-\log_2$ of the probability that a random mutation will change A to B . Using AIT [14, 15, 16], we see that this is nearly $H(B|A)$, the size in bits of the smallest self-delimiting program that takes A as input and produces B as output.⁴ More precisely,

$$H(B|A) = -\log_2 P(B|A) + O(1) = -\log_2 \left[\sum_{U(p|A)=B} 2^{-|p|} \right] + O(1). \quad (1)$$

Here $|p|$ denotes the size in bits of the program p , and $U(p|A)$ denotes the output produced by running p given input A on the computer U until p halts.

³The total probability of mutations is actually < 1 , so that each time we pick a mutation at random, there is a fixed probability that we will get the *null mutation* $M(A) = A$, which always fails.

⁴Similarly, $H(B)$ denotes the size in bits of the smallest self-delimiting program for B that is *not* given A . $H(B)$ is called the *complexity* of B , and $H(B|A)$ is the *relative complexity* of B given A .

The definition of $H(B|A)$ that we employ here is somewhat different from the one that is used in AIT: a mutation is given A directly, it is not given a minimum-size program for A . Nevertheless, (1) holds [14].

Interpreting (1) in words, it is nearly the same to consider the *simplest* mutation from A to B , which is $H(B|A)$ bits in size and has probability $2^{-H(B|A)}$, as to sum the probability over *all* the mutations that carry A into B .

Note that this distance measure is not symmetric. For example, it is easy to change (X, Y) into Y , but not vice versa.

4.6 Hidden Use of Oracles

There are two hidden assumptions here. First of all, we need to use an oracle to compare the fitness of an organism A with that of a mutated organism A' . This is because a mutated program may not halt and thus never produces a natural number. Once we know that the original organism A and the mutated organism A' both halt, then we can run them to see what they calculate and which is fitter.

In the case of fast-growing computable functions, an oracle is definitely needed to see if one grows faster than another; this cannot be determined by running the primitive recursive functions [29] calculated by the FORTRAN-like programs that we will study later, in Section 8.

Just as oracles would be needed to actually find fitter organisms, they are also necessary because a random mutation may never halt and produce a mutated organism. So to actually apply our random mutations to organisms we would need to use an oracle in order to avoid non-terminating mutations.

5 Model A (Naming Integers) Exhaustive Search

5.1 The Busy Beaver Function

The first step in this metabiological approach is to measure the rate of evolution. To do that, we introduce this version of the Busy Beaver function:

$BB(N)$ = the biggest natural number named by a $\leq N$ -bit program.

More formally,

$$BB(N) = \max_{H(k) \leq N} k.$$

Here the program-size *complexity* or the algorithmic *information content* $H(k)$ of k is the size in bits of the smallest self-delimiting program p without input for calculating k :

$$H(k) = \min_{U(p)=k} |p|.$$

Here again $|p|$ denotes the size in bits of p , and $U(p)$ denotes the output produced by running the program p on the computer U until p halts.

5.2 Proof of Theorem 1 (Exhaustive Search)

Now, for the sake of definiteness, let's start with the trivial program that directly outputs the positive integer 1, and apply mutations at random.⁵ Let's define the *mutation time* to be n if we have tried n mutations, and the *organism time* to be n if there are n successive organisms of increasing fitness so far in our infinite random walk.

From AIT [14] we know that there is an $N + O(1)$ -bit mutation that ignores its input and produces as output a $\leq N$ -bit program that calculates $BB(N)$. This mutation M has probability $2^{-N+O(1)}$ and on the average, it will occur at random every $2^{N+O(1)}$ times a random mutation is tried. Therefore:

Theorem 1 *The fitness of our organism will reach $BB(N)$ by mutation time 2^N . In other words, we will achieve N bits of biological/mathematical creativity by time 2^N . Each successive bit of creativity takes twice as long as the previous bit did.*⁶

More precisely, the probability that this should fail to happen, the probability that M has *not* been tried by time 2^N , is

$$\left(1 - \frac{1}{2^N}\right)^{2^N} \rightarrow e^{-1} \approx \frac{1}{2.7} < \frac{1}{2}.$$

And the probability that it will fail to happen by mutation time $K2^N$ is $< 1/2^K$.

This is the worst that evolution can do. It is the fitness that organisms will achieve if we are employing *exhaustive search* on the space of all possible organisms. Actual biological evolution is not at all like that. The human genome has 3×10^9 bases, but in the mere 4×10^9 years of life on this planet only a tiny fraction of the total enormous number $4^{3 \times 10^9}$ of sequences of 3×10^9 bases can have been tried. In other words, evolution is not *ergodic*.

6 Model A (Naming Integers) Intelligent Design

6.1 Another Busy Beaver Function

If we could choose our mutations intelligently, evolution would be much more rapid. Let's use the halting probability Ω [19] to show just how rapid. First we define a slightly different Busy Beaver function BB' based on Ω . Consider a fixed recursive/computable enumeration $\{p_i : i = 0, 1, 2 \dots\}$ without repetitions of all the programs without input that halt when run on U . Thus

$$0 < \Omega = \Omega_U = \sum_i 2^{-|p_i|} < 1 \quad (2)$$

⁵The choice of initial organism is actually unimportant.

⁶Instead of *bits of creativity* one could perhaps refer to *bits of inspiration*; said inspiration of course is ultimately coming through/from our oracle, which keeps us from getting stuck on non-terminating programs.

and we get the following sequence $\Omega_0 = 0 < \Omega_1 < \Omega_2 \dots$ of lower bounds on Ω :

$$\Omega_N = \sum_{i < N} 2^{-|p_i|}. \quad (3)$$

In (2) and (3) $|p|$ denotes the size in bits of p , as before.

We define $\text{BB}'(K)$ to be the least N for which the first K bits of the base-two numerical value of Ω_N are correct, i.e., the same as the first K bits of the numerical value of Ω . $\text{BB}'(K)$ exists because we know from AIT [14] that Ω is irrational, so $\Omega = .010000$ is impossible and there is no danger that Ω_N will be of the form $.0011111$ with 1's forever.

Note that **BB and BB' are approximately equal**. For we can calculate $\text{BB}'(N)$ if we are given N and the first N bits of Ω . Therefore

$$\text{BB}'(N) \leq \text{BB}(N + H(N) + c) = \text{BB}(N + O(\log N)).$$

Furthermore, if we knew N and any $M \geq \text{BB}'(N)$, we could calculate the string ω of the first N bits of Ω , which according to AIT [14] has complexity $H(\omega) > N - c'$, so

$$N - c' < H(\omega) \leq H(N) + H(M) + c''.$$

Therefore $\text{BB}'(N)$ and all greater than or equal numbers M have complexity $H(M) > N - H(N) - c' - c''$, so $\text{BB}'(N)$ must be greater than the biggest number M_0 with complexity $H(M_0) \leq N - H(N) - c' - c''$. Therefore

$$\text{BB}'(N) > \text{BB}(N - H(N) - c' - c'') = \text{BB}(N + O(\log N)).$$

6.2 Improving Lower Bounds on Ω

Our model consists of arbitrary mutation computer programs operating on arbitrary organism computer programs. To analyze the behavior of this system (Model A), however, we shall focus on a select subset: Our organisms are lower bounds on Ω , and our mutations increase these lower bounds.

We are going to use these same organisms and mutations to analyze both intelligent design (Section 6.3) and cumulative evolution at random (Section 7). Think of Section 6.3 versus Section 7 as counterpoint.

6.2.1 Organism P_ρ — Lower Bound ρ on Ω

Now we use a bit string ρ to represent a dyadic rational number in $[0, 2) = \{0 \leq x < 2\}$; ρ consists of the base-two units “digit” followed by the base-two expansion of the fractional part of this rational number.

There is a self-delimiting prefix π_Ω that given a bit string ρ that is a lower bound on Ω , calculates the first N such that $\Omega > \Omega_N \geq \rho$, where Ω_N is defined as in (3).⁷ If we concatenate the prefix π_Ω with the string of bits ρ , and insert $0^{|\rho|}1$

⁷That $\rho \neq \Omega$ follows from the fact that Ω is irrational.

in front of ρ in order to make everything self-delimiting, we obtain a program P_ρ for this N .

We will now analyze the behavior of Model A by using these organisms of the form

$$P_\rho = \pi_\Omega 0^{|\rho|} 1\rho. \quad (4)$$

To repeat, the output of P_ρ , and therefore its fitness ϕ_{P_ρ} , is determined as follows:

$$U(P_\rho) = \text{the first } N \text{ for which } \sum_{i < N} 2^{-|p_i|} = \Omega_N \geq \rho. \quad (5)$$

This fitness will be $\geq \text{BB}'(K)$ if $\rho < \Omega$ and the first K bits of ρ are the correct base-two numerical value of Ω . P_ρ will fail to halt if $\rho > \Omega$.⁸

6.2.2 Mutation M_k — Lower Bound ρ on Ω Increased by 2^{-k}

Consider the mutations M_k that do the following. First of all, M_k computes the fitness ϕ of the current organism A by running A to determine the integer $\phi = \phi_A$ that A names. **All that M_k takes from A is its fitness ϕ_A .** Then M_k computes the corresponding lower bound on Ω :

$$\rho = \sum_{i < \phi} 2^{-|p_i|} = \Omega_\phi.$$

Here $\{p_i\}$ is the standard enumeration of all the programs that halt when run on U that we employed in Section 6.1. Then M_k increments the lower bound ρ on Ω by 2^{-k} :

$$\rho' = \rho + 2^{-k}.$$

In this way M_k obtains the mutated program

$$A' = P_{\rho'}.$$

A' will fail to halt if $\rho' > \Omega$. If A' does halt, then $A' = M_k(A) = P_{\rho'}$ will have fitness N (see (5)) greater than $\phi_A = \phi$ because $\rho' > \rho = \Omega_\phi$, so more halting programs are included in the sum (3) for Ω_N , which therefore has been extended farther:

$$[\Omega_N \geq \rho' > \rho = \Omega_\phi] \implies [N > \phi].$$

**Therefore if $\Omega > \rho' = \rho + 2^{-k}$, then M_k increases the fitness of A .
If $\rho' > \Omega$, then $P_{\rho'} = M_k(A)$ never halts and is totally unfit.**

6.3 Proof of Theorem 2 (Intelligent Design)

Please note that in this toy world, the “intelligent designer” is the author of this paper, who chooses the mutations optimally in order to get his creatures to evolve.

⁸That $\rho \neq \Omega$ follows from the fact that Ω is irrational.

Let's now start with the computer program P_ρ with $\rho = 0$. In other words, we start with a lower bound on Ω of zero.

Then for $k = 1, 2, 3 \dots$ we try applying M_k to P_ρ . The mutated organism $P_{\rho'} = M_k(P_\rho)$ will either fail to halt, or it will have higher fitness than our previous organism and will replace it. Note that in general $\rho' \neq \rho + 2^{-k}$, although it could conceivably have that value. M_k will from P_ρ take only its fitness, which is the first N such that $\Omega_N \geq \rho$.

$$\rho' = \Omega_N + 2^{-k} \geq \rho + 2^{-k}.$$

So ρ' is actually equal to a lower bound on Ω , Ω_N , plus 2^{-k} . Thus M_k will attempt to increase a lower bound on Ω , Ω_N , by 2^{-k} . M_k will succeed if $\Omega > \rho'$. M_k will fail if $\rho' > \Omega$. This is the situation at the end of stage k . Then we increment k and repeat. The lower bounds on Ω will get higher and higher.

More formally, let $O_0 = P_\rho$ with $\rho = 0$. And for $k \geq 1$ let

$$O_k = \begin{cases} O_{k-1} & \text{if } M_k \text{ fails,} \\ M_k(O_{k-1}) & \text{if } M_k \text{ succeeds.} \end{cases}$$

Each O_k is a program of the form P_ρ with $\Omega > \rho$.

At the end of stage k in this process the first k bits of ρ will be exactly the same as the first k bits of Ω , because at that point all together we have tried summing $1/2 + 1/4 + 1/8 \dots + 1/2^k$ to ρ . **In essence, we are using an oracle to determine the value of Ω by successive interval halving.**⁹

In other words, at the end of stage k the first k bits of ρ in O_k are correct. Hence:

Theorem 2 *By picking our mutations intelligently rather than at random, we obtain a sequence O_N of software organisms with non-decreasing fitness¹⁰ for which the fitness of each organism is $\geq BB'(N)$. In other words, we will achieve N bits of biological/mathematical creativity in mutation time linear in N . Each successive bit of creativity takes about as long as the previous bit did.*

However, successive mutations must be tried at random in our evolution model; they cannot be chosen deliberately. We see in these two theorems two extremes: Theorem 1, brainless exhaustive search, and Theorem 2, intelligent design. What can real, random evolution actually achieve? We shall see that the answer is closer to Theorem 2 than to Theorem 1. We will achieve fitness $BB'(N)$ in time roughly order of N^2 . In other words, each successive bit of creativity takes an amount of time which increases linearly in the number of bits.

⁹That this works is easy to see visually. Think of the unit interval drawn vertically, with 0 below and 1 above. The intervals are being pushed up after being halved, but it is still the case that Ω remains inside each halved interval, even after it has been pushed up.

¹⁰Note that this is actually a legitimate fitness increasing (non-random) walk because the fitness increases each time that O_N changes, i.e., each time that $O_{N+1} \neq O_N$.

Open Problem 1 *Is this the best that can be done by picking the mutations intelligently rather than at random? Or can creativity be even faster than linear? Does each use of the oracle yield only one bit of creativity?* ¹¹

Open Problem 2 *In Theorem 2 how fast does the size in bits of the organism O_N grow? By using entirely different mutations intelligently, would it be possible to have the size in bits of the organism O_N grow linearly, or, alternatively, for the mutation distance between O_N and O_{N+1} to be bounded, and still achieve the same rapid growth in fitness?*

Open Problem 3 *In Theorem 2 how many different organisms will there be by mutation time N ? I.e., on the average how fast does organism time grow as a function of mutation time?*

7 Model A (Naming Integers) Cumulative Evolution at Random

Now we shall achieve what Theorem 2 achieved by intelligent design, by using randomness instead. Since the order of our mutations will be random, not intelligent, **there will be some duplication of effort and creativity is delayed, but not overmuch.**

In other words, instead of using the mutations M_k in a predetermined order, they shall be picked at random, and also mixed together with other mutations that increase the fitness.

As you will recall (Section 6.2), a larger and larger positive integer is equivalent to a better and better lower bound on Ω . That will be our clock, our memory. We will again be evolving better and better lower bounds ρ on Ω and we shall make use of the organisms P_ρ as before ((4), Section 6.2.1). We will also use again the mutations M_k of Section 6.2.2.

Let's now study the behavior of the random walk in Model A if we start with an arbitrary program A that has a fitness, for example, the program that is the constant 0, and apply mutations to it at random, according to the probability measure on mutations determined by AIT [14], namely that M has probability $2^{-H(M)}$.¹² So with probability one, **every mutation will be tried infinitely often**; M will be tried roughly every $2^{H(M)}$ mutation times.

At any given point in this random walk, we can measure our progress to Ω by the fitness $\phi = \phi_A$ of our current organism A and the corresponding lower bound $\Omega_\phi = \Omega_{\phi_A}$ on Ω . Since the fitness ϕ can only increase, the lower bound Ω_ϕ can only get better.

In our analysis of what will happen we focus on the mutations M_k ; other mutations will have no effect on the analysis. They are harmless and can be

¹¹Yes, only one bit of creativity, otherwise Ω would be compressible. In fact, the sequence of oracle replies must be incompressible.

¹²This is a convenient lower bound on the probability of a mutation. A more precise value for the probability of jumping from A to A' is $2^{-H(A'|A)}$.

mixed in together with the M_k . By increasing the fitness, they can only make Ω_ϕ converge to Ω more quickly.

We also need a *new* mutation M^* . M^* doesn't get us much closer to Ω , it just makes sure that our random walk will contain infinitely many of the programs P_ρ . M^* will be tried roughly periodically during our random walk. M^* takes the current lower bound $\Omega_\phi = \Omega_{\phi_A}$ on Ω , and produces

$$A' = M^*(A) = P_{\Omega_{1+\phi_A}}.$$

A' has fitness 1 greater than the fitness of A and thus mutation M^* will always succeed, and this keeps lots of organisms of the form P_ρ in our random walk.

Let's now return to the mutations M_k , each of which will also have to be tried infinitely often in the course of our random walk.

The mutation M_k will either have no effect because $M_k(A)$ fails to halt, which means that we are less than 2^{-k} away from Ω , that is, Ω_{ϕ_A} is less than 2^{-k} away from Ω , or M_k will have the effect of incrementing our lower bound Ω_{ϕ_A} on Ω by 2^{-k} . As more and more of these mutations M_k are tried at random, eventually, purely by chance, more and more of the beginning of Ω_{ϕ_A} will become correct (the same as the initial bits of Ω). Meanwhile, the fitness ϕ_A will increase enormously, passing $BB'(n)$ as soon as the first n bits of Ω_{ϕ_A} are correct. And soon afterwards, M^* will package this in an organism $A' = P_{\Omega_{1+\phi_A}}$.

How long will it take for all this to happen? I.e., how long will it take to try the M_k for $k = 1, 2, 3, \dots, n$ and then try M^* ? We have

$$H(M_k) \leq H(k) + c.$$

Therefore mutation M_k has probability

$$\geq 2^{-H(k)-c} > \frac{1}{c'k(\log k)^{1+\epsilon}} \quad (6)$$

since

$$\sum_k \frac{1}{k(\log k)^{1+\epsilon}}$$

converges.¹³ The mutation M_k will be tried in time proportional to 1 over the probability of its being tried, which by (6) is approximately upper bounded by

$$\xi(k) = c''k(\log k)^{1+\epsilon}. \quad (7)$$

On the average, from what point on will the first n bits of $\Omega_\phi = \Omega_{\phi_A}$ be the same as the first n bits of Ω ? We can be sure this will happen if we first try M_1 , then afterwards M_2 , then M_3 , etc. through M_n , in that order. Note that if these mutations are tried in the wrong order, they will not have the desired effect. But they will do no harm either, and eventually will *also* be tried in the correct order. Note that it is conceivable that none of these M_k actually succeed, because of the other random mutations that were in the mix, in the

¹³We are using here one of the basic theorems of AIT [14].

melee. These other mutations may *already* have pushed us within 2^{-k} of Ω . So these M_k don't have to succeed, they just have to be tried. Then M^* will make sure that we get an organism of the form P_ρ with at least n bits of ρ correct.

Hence:

$$\begin{aligned}
&\text{Expected time to try } M_1 &\leq \xi(1) \\
&\text{Expected time to then afterwards try } M_2 &\leq \xi(2) \\
&\text{Expected time to then afterwards try } M_3 &\leq \xi(3) \\
&&\dots \\
&\text{Expected time to then afterwards try } M_n &\leq \xi(n) \\
&\text{Expected time to then afterwards try } M^* &\leq c''' \\
\therefore \text{Expected time to try } M_1, M_2, M_3 \dots M_n, M^* \text{ in order} &\leq \sum_{k \leq n} \xi(k) + c'''
\end{aligned}$$

Using (7), we see that this is our extremely rough “ball-park” estimate on a mutation time sufficiently big for the first n bits of ρ in $P_\rho = M^*(A)$ to be the correct bits of Ω :

$$\sum_{k \leq n} \xi(k) + c''' = \sum_{k \leq n} c'' k (\log k)^{1+\epsilon} + c''' = O(n^2 (\log n)^{1+\epsilon}). \quad (8)$$

Hence we expect that in time $O(n^2 (\log n)^{1+\epsilon})$ our random walk will include an organism P_ρ in which the first n bits of ρ are correct, and so P_ρ will compute a positive integer $\geq BB'(n)$, and thus at this time the fitness will have to be at least that big:

Theorem 3 *In Model A with random mutations, the fitness of the organisms $P_\rho = M^*(A)$ will reach $BB'(N)$ by mutation time roughly N^2 .*

Note that since the bits of ρ in the organisms $P_\rho = M^*(A)$ are becoming better and better lower bounds on Ω , these organisms in effect contain their evolutionary history. **In Model A, evolution is cumulative, it does not start over from scratch as in exhaustive search.**

It should be emphasized that in the course of such a hill-climbing random walk, with probability one every possible mutation will be tried infinitely often. However the mutations M_k will immediately recover from perturbations and set the evolution back on course. In a sense the system is *self-organizing* and *self-repairing*. Similarly, the initial organism is irrelevant.

Also note that with probability one the time history or evolutionary pathway (i.e., the random walk in Model A) will quickly grow better and better approximations to *all possible* halting probabilities $\Omega_{U'}$ (see (2)) determined by *any* optimal universal self-delimiting binary computer U' , not just for our original U . Furthermore, some mutations will periodically convert our organism into a numerical constant for its fitness ϕ , and there will even be arbitrarily long chains of successive numerical constant organisms $\phi, \phi + 1, \phi + 2 \dots$. The microstructure and fluctuations that will occur with probability one are quite varied and should perhaps be studied in detail to unravel the full zoo of organisms and their interconnections; this is in effect a kind of miniature mathematical ecology.

Open Problem 4 Study this *mathematical ecology*.

Open Problem 5 Improve the estimate (8) and get a better upper bound on the expected time it will take to try M_1, M_2, M_3 through M_n and M^* in that order. Besides the mean, what is the variance?

Open Problem 6 *Separate* random evolution and intelligent design: We have shown that random evolution is fast, but can you prove that it cannot be as fast as intelligent design? I.e., we have a lower bound on the speed of random evolution, and now we also need an upper bound. This is probably easier to do if we only consider random mutations M_k and keep other mutations from mixing in.

Open Problem 7 In Theorem 3 how fast does the size in bits of the organism P_ρ grow? Is it possible to have the size in bits of the organism P_ρ grow linearly and still achieve the same rapid growth in fitness?

Open Problem 8 It is interesting to think of Model A as a conventional random walk and to study the average mutation distance between an organism A and its successor A' , its second successor A'' , etc. In organism time Δt how far will we get from A on the average? What will the variance be?

8 Model B (Naming Functions)

Let's now consider Model B. Why study Model B? Because hierarchical structure is a conspicuous feature of actual biological organisms, but it is impossible to prove that such structure must emerge by random evolution in Model A.

Why not? Because the programming language used by the organisms in Model A is so powerful that all structure in the programs can be hidden. Consider the programs P_ρ defined in Section 6.2.1 and used to prove Theorems 2 and 3. As we saw in Theorem 3, these programs P_ρ evolve without limit at random. However, P_ρ consists of a fixed prefix π_Ω followed by a lower bound on Ω, ρ , and what evolves is the lower bound ρ , *data* which has no visible hierarchical structure, not the prefix π_Ω , *code* which has fixed, unevolving, hierarchical structure.

So in Model A it is impossible to prove that hierarchical structure will emerge and increase in depth. To be able to do this we must utilize a less powerful programming language, one that is not universal and in which the hierarchical structure cannot be hidden: the Meyer-Ritchie LOOP language [28].

We will show that the nesting depth of LOOP programs will increase without limit, due to random mutations. This also provides a much more concrete example of evolution than is furnished by our main model, Model A.

Now for the details.

We study the evolution of functions $f(x)$ of a single integer argument x ; faster growing functions are taken to be fitter. More precisely, if $f(x)$ and $g(x)$ are two such functions, f is fitter than g iff $g/f \rightarrow 0$ as $x \rightarrow \infty$. We use an oracle

to decide if $A' = M(A)$ is fitter than A ; if not, A is not replaced by A' .¹⁴ The programming language we are using has the advantage that program structure cannot be hidden. It's a programming language that is powerful enough to program any primitive recursive function [29], but it's not a universal programming language.

To give a concrete example of hierarchical evolution, we use the extremely simple Meyer-Ritchie LOOP programming language, containing only assignment, addition by 1, do loops, and no conditional statements or subroutines. All variables are natural numbers, non-negative integers. Here is an example of a program written in this language:

```
// Exponential: 2 to the Nth power
// with only two nested do loops!
function(N) // Parameter must be called N.
  M = 1
  //
  do N times
    M2 = 0
    // M2 = 2 * M
    do M times
      M2 = M2 + 1
      M2 = M2 + 1
    end do
    M = M2
  end do
  // Return M = 2 to the Nth power.
  return_value = M
  // Last line of function must
  // always set return_value.
end function
```

More generally, let's start with $f_0(x) = 2x$:

```
function(N) // f_0(N)
  M = 0
  // M = 2 * N
  do N times
    M = M + 1
    M = M + 1
  end do
  return_value = M
end function // end f_0(N)
```

Note that the nesting depth of f_0 is 1.

¹⁴An oracle is needed in order to decide whether $g(x)/f(x) \rightarrow 0$ as $x \rightarrow \infty$ and also to avoid mutations M that never produce an $A' = M(A)$. Furthermore, if a mutation produces a syntactically invalid LOOP program A' , A' does not replace A .

And given a program for the function f_k , here is how we program

$$f_{k+1}(x) = f_k^x(2) \tag{9}$$

by increasing the nesting depth of the program for f_k by 1:

```
function(N) // f_(k+1)(N)
  M = 2
  // do M = f_k(M) N times
  do N times
    N_ = M
    // Insert program for f_k here
    // with "function" and "end function"
    // stripped and all variable names
    // renamed to variable name_
    M = return_value_
  end do
  return_value = M
end function // end f_(k+1)(N)
```

So following (9) we now have programs for

$$f_0(x) = 2x, \quad f_1(x) = 2^x, \quad f_2(x) = 2^{2^{\dots}} \text{ with } x \text{ 2's} \dots$$

Note that a program in this language which has nesting depth 0 (no do loops) can only calculate a function of the form $(x + \text{a constant})$, and that the depth 1 function $f_0(x) = 2x$ grows faster than all of these depth 0 functions. More generally, it can be proven by induction [29] that a program in this language with do loop nesting depth $\leq k$ defines functions that grow more slowly than f_k , which is defined by a depth $k + 1$ LOOP program. This is the basic theorem of Meyer and Ritchie [28] classifying the primitive recursive functions according to their rates of growth.

Now consider the mutation M that examines a software organism A written in this LOOP language to determine its nesting depth n , and then replaces A by $A' = f_n(x)$, a function that grows faster than any LOOP function with depth $\leq n$. Mutation M will be tried at random with probability $\geq 2^{-H(M)}$. And so:

Theorem 4 *In Model B, the nesting depth of a LOOP function will increase by 1 roughly periodically, with an estimated mutation time of $2^{H(M)}$ between successive increments. Once mutation M increases the nesting depth, it will remain greater than or equal to that increased depth, because no LOOP function with smaller nesting depth can grow as fast.*

Note that this theorem works because the nesting depth of a primitive recursive function is used as a clock; it gives Model B memory that can be used by intelligent mutations like M .

Open Problem 9 *In the proof of Theorem 4, is the mutation M primitive recursive, and if so, what is its LOOP nesting depth?*

Open Problem 10 *M can actually increase the nesting depth extremely fast. Study this.*

Open Problem 11 *Formulate a version of Theorem 4 in terms of subroutine nesting instead of do loop nesting. What is a good computer programming language to use for this?*

9 Remarks on Model C (Naming Ordinals)

Now let's briefly turn to programs that compute constructive Cantor ordinal numbers α [27]. From a biological point of view, the evolution of ordinals is piquant, because they certainly exhibit a great deal of hierarchical structure. Not, in effect, as we showed in Section 8 must occur in the genotype; here it is *automatically* present in the phenotype.

Ordinals also seem like an excellent choice for an evolutionary model because of their fundamental role in mathematics¹⁵ and because of the mystique associated with naming large ordinals, a problem which can utilize an unlimited amount of mathematical creativity [26, 27]. Conventional ordinal notations can only handle an initial segment of the constructive ordinals.

However there are two fundamentally different ways [27] to use algorithms to name *all* such ordinals α :

- An ordinal is a program that given two positive integers, tells us which is less than the other in a well-ordering of the positive integers with order type α .
- An ordinal α is a program for obtaining that ordinal from below: If it is a successor ordinal, as $\beta + 1$; if it is a limit ordinal, as the limit of a fundamental sequence β_k ($k = 0, 1, 2 \dots$).

This yields two different definitions of the algorithmic information content or program-size complexity of a constructive ordinal:

$H(\alpha)$ = the size in bits of the smallest self-delimiting program
for calculating α .

We can now define this beautiful new version of the Busy Beaver function:

$$\text{BB}_{ord}(N) = \max_{H(\alpha) \leq N} \alpha.$$

In order to make programs for ordinals α evolve, we now need to use a very sophisticated oracle, one that can determine if a program computes an ordinal and, given two such programs, can also determine if one of these ordinals is less than the other. Assuming such an oracle, we get the following version of Theorem 1, merely by using brainless exhaustive search:

¹⁵As an illustration of this, ordinals may be used to extend the function hierarchy f_k of Section 8 to transfinite k . For example, $f_\omega(x) = f_x(x)$, $f_{\omega+1}(x) = f_\omega^x(2)$, $f_{\omega+2}(x) = f_{\omega+1}^x(2)$ \dots $f_{\omega \times 2}(x) = f_{\omega+x}(x)$, etc., an extension of (9).

Theorem 5 *The fitness of our ordinal organism α will reach $BB_{ord}(N)$ by mutation time 2^N .*

Can we do better than this? The problem is to determine if there is some kind of Ω number or other way to compress information about constructive ordinals so that we can improve on Theorem 5 by proving that evolution will probably reach $BB_{ord}(N)$ in an amount of time which *does not* grow exponentially.

We suspect that Model C may be an example of a case in which *cumulative evolution at random does not occur*. On the other hand, we are given an extremely powerful oracle; maybe it is possible to take advantage of that. The problem is open.

Open Problem 12 *Improve on Theorem 5 or show that no improvement is possible.*

10 Conclusion

At this point we should look back and ask why this all worked. Mainly for the following reason: We used an extremely rich space of possible mutations, one that possess a natural probability distribution: the space of all possible self-delimiting programs studied by AIT [14]. But the use of such powerful mutational mechanisms raises a number of issues.

Presumably DNA is a universal programming language, but how sophisticated can mutations be in actual biological organisms? In this connection, note that evo-devo views DNA as software for constructing the embryo, and that the change from single-celled to multicellular organisms is roughly like taking a main program and making it into a subroutine, which is a fairly high-level mutation. Could this be the reason that it took so long—on the order of 10^9 years—for this to happen?¹⁶

The issue of *balance* between the power of the organisms and the power of the mutations is an important one. In the current version of the theory, both have equal power, but as a matter of aesthetics it would be bad form for a proof to overemphasize the mutations at the expense of the organisms. In future versions of the theory perhaps it will be desirable to limit the power of mutations in some manner by fiat.

In this connection, note that there are two uses of oracles in this theory, one to decide which of two organisms is fitter, and another to eliminate non-terminating mutations. It is perfectly fine for a proof to be based on taking advantage of the oracle for organisms, but taking advantage of the oracle for mutations is questionable.

We have by no means presented in this paper a mathematical theory of evolution and biological creativity *comme il faut*. But at this point in time we believe that metabiology is still a possible contender for such a theory. The ultimate goal must be to find in the Platonic world of mathematical ideas that ideal

¹⁶During most of the history of the earth, life was unicellular.

model of evolution by natural selection which real, messy biological evolution can but approach asymptotically in the limit from below.

We thank Prof. Cristian Calude of the University of Auckland for reading a draft of this paper, for his helpful comments, and for providing the paper by Meyer and Ritchie [28].

Appendix. AIT in a Nutshell

Programming languages are commonly *universal*, that is to say, capable of expressing essentially any algorithm.

In order to be able to combine subroutines, i.e., for algorithmic information to be *subadditive*,

$$\begin{aligned} & \text{size of program to calculate } x \text{ and } y \\ & \leq \text{size of program to calculate } x \\ & \quad + \text{size of program to calculate } y, \end{aligned}$$

it is important that programs be *self-delimiting*. This means that the universal computer U reads a program bit by bit as required and there is no special delimiter to mark the end of the program; the computer must decide by itself where to stop reading.

More precisely, if programs are self-delimiting we have

$$H(x, y) \leq H(x) + H(y) + c,$$

where $H(\dots)$ denotes the size in bits of the smallest program for U to calculate \dots , and c is the number of bits in the main program that reads and executes the subroutine for x followed by the subroutine for y .

Besides giving us subadditivity, the fact that programs are self-delimiting also enables us to talk about that probability $P(x)$ that a program that is generated at random will compute x when run on U .

Let's now consider how expressive different programming languages can be. Given a particular programming language U , two important things to consider are the *program-size complexity* $H(x)$ as a function of x , and the corresponding *algorithmic probability* $P(x)$ that a program whose bits are chosen using independent tosses of a fair coin will compute x .

We are thus led to select a subset of the universal languages that minimize H and maximize P ; one way to define such a language is to consider a universal computer U that runs self-delimiting binary computer programs $\pi_C p$ defined as follows:

$$U(\pi_C p) = C(p).$$

In other words, the result of running on U the program consisting of the prefix π_C followed by the program p , is the same as the result of running p on the computer C . The prefix π_C tells U which computer C to simulate.

Any two such maximally expressive universal languages U and V will necessarily have

$$|H_U(x) - H_V(x)| \leq c$$

and

$$P_U(x) \geq P_V(x) \times 2^{-c}, \quad P_V(x) \geq P_U(x) \times 2^{-c}.$$

It is in this precise sense that such a universal U minimizes H and maximizes P .

For such languages U it will be the case that

$$H(x) = -\log_2 P(x) + O(1),$$

which means that most of the probability of calculating x is concentrated on the minimum-size program for doing this, which is therefore essentially unique. $O(1)$ means that the difference between the two sides of the equation is order of unity, i.e., bounded by a constant.

Furthermore, we have

$$H(x, y) = H(x) + H(y|x) + O(1).$$

Here $H(y|x)$ is the size of the smallest program to calculate y from x .¹⁷ This tells us that essentially the best way to calculate x and y is to calculate x and then calculate y from x . In other words, the *joint complexity* of x and y is essentially the same as the *absolute complexity* of x added to the *relative complexity* of y given x .

This decomposition of the joint complexity as a sum of absolute and relative complexities implies that the *mutual information* content

$$H(x : y) \equiv H(x) + H(y) - H(x, y),$$

which is the extent to which it is easier to compute x and y together rather than separately, has the property that

$$H(x : y) = H(x) - H(x|y) + O(1) = H(y) - H(y|x) + O(1).$$

In other words, $H(x : y)$ is also the extent to which knowing y helps us to know x and vice versa.

Last but not least, using such a maximally expressive U we can define the *halting probability* Ω , for example as follows:

$$\Omega = \sum 2^{-|p|}$$

summed over all programs p that halt when run on U , or alternatively

$$\Omega' = \sum 2^{-H(n)}$$

summed over all positive integers n , which has a slightly different numerical value but essentially the same paradoxical properties.

What are these properties? Ω is a form of concentrated mathematical creativity, or, alternatively, a particularly economical Turing oracle for the halting

¹⁷It is crucial that we are not given x directly. Instead we are given a minimum-size program for x .

problem, because knowing n bits of the dyadic expansion of Ω enables one to solve the halting problem for all programs p which compute a positive integer that are up to n bits in size. It follows that the bits of the dyadic expansion of Ω are irreducible mathematical information; they cannot be compressed into a theory smaller than they are.¹⁸

From a philosophical point of view, however, the most striking thing about Ω is that it provides a perfect simulation in pure mathematics, where all truths are necessary truths, of contingent, accidental truths—i.e., of truths such as historical facts or biological frozen accidents.

Furthermore, Ω opens a door for us from mathematics to biology. The halting probability Ω contains infinite irreducible complexity and in a sense shows that pure mathematics is even more biological than biology itself, which merely contains extremely large finite complexity. For each bit of the dyadic expansion of Ω is one bit of independent, irreducible mathematical information, while the human genome is merely 3×10^9 bases = 6×10^9 bits of information.

References

- [1] D. Berlinski, *The Devil's Delusion*, Crown Forum, 2008.
- [2] S. J. Gould, *Wonderful Life*, Norton, 1990.
- [3] N. Shubin, *Your Inner Fish*, Pantheon, 2008.
- [4] M. Mitchell, *Complexity*, Oxford University Press, 2009.
- [5] J. Fodor, M. Piattelli-Palmarini, *What Darwin Got Wrong*, Farrar, Straus and Giroux, 2010.
- [6] S. C. Meyer, *Signature in the Cell*, HarperOne, 2009.
- [7] J. Maynard Smith, *Shaping Life*, Yale University Press, 1999.
- [8] J. Maynard Smith, E. Szathmary, *The Origins of Life*, Oxford University Press, 1999; *The Major Transitions in Evolution*, Oxford University Press, 1997.
- [9] J. P. Crutchfield, O. Gornerup, “Objects that make objects: The population dynamics of structural complexity,” *Journal of the Royal Society Interface* **3** (2006), pp. 345–349.
- [10] G. J. Chaitin, “Evolution of mutating software,” *EATCS Bulletin* **97** (February 2009), pp. 157–164.
- [11] G. J. Chaitin, *Mathematics, Complexity and Philosophy*, Midas, in press. (See Chapter 3, “Algorithmic Information as a Fundamental Concept in Physics, Mathematics and Biology.”)

¹⁸More precisely, it takes a formal axiomatic theory of complexity $\geq n - c$ (one requiring a $\geq n - c$ bit program to enumerate all its theorems) to enable us to determine n bits of Ω .

- [12] G. J. Chaitin, “Metaphysics, metamathematics and metabiology,” in P. García, A. Massolo, *Epistemología e Historia de la Ciencia: Selección de Trabajos de las XX Jornadas*, **16** (2010), Facultad de Filosofía y Humanidades, Universidad Nacional de Córdoba, pp. 178–187. Also in *APA Newsletter on Philosophy and Computers* **10**, No. 1 (Fall 2010), pp. 7–11, and in H. Zenil, *Randomness Through Computation*, World Scientific, 2011, pp. 93–103.
- [13] G. Chaitin, *Proving Darwin: Making Biology Mathematical*, Pantheon, to appear.
- [14] G. J. Chaitin, “A theory of program size formally identical to information theory,” *J. ACM* **22** (1975), pp. 329–340.
- [15] G. J. Chaitin, *Algorithmic Information Theory*, Cambridge University Press, 1987.
- [16] G. J. Chaitin, *Exploring Randomness*, Springer, 2001.
- [17] C. S. Calude, *Information and Randomness*, Springer-Verlag, 2002.
- [18] M. Li, P. M. B. Vitányi, *An Introduction to Kolmogorov Complexity and Its Applications*, Springer, 2008.
- [19] C. Calude, G. Chaitin, “What is a halting probability?,” *AMS Notices* **57** (2010), pp. 236–237.
- [20] H. Steinhaus, *Mathematical Snapshots*, Oxford University Press, 1969, pp. 29–30.
- [21] D. E. Knuth, “Mathematics and computer science: Coping with finiteness,” *Science* **194** (1976), pp. 1235–1242.
- [22] A. Hodges, *One to Nine*, Norton, 2008, pp. 246–249; M. Davis, *The Universal Computer*, Norton, 2000, pp. 169, 235.
- [23] G. J. Chaitin, “Computing the Busy Beaver function,” in T. M. Cover, B. Gopinath, *Open Problems in Communication and Computation*, Springer, 1987, pp. 108–112.
- [24] G. H. Hardy, *Orders of Infinity*, Cambridge University Press, 1910. (See Theorem of Paul du Bois-Reymond, p. 8.)
- [25] D. Hilbert, “On the infinite,” in J. van Heijenoort, *From Frege to Gödel*, Harvard University Press, 1967, pp. 367–392.
- [26] J. Stillwell, *Roads to Infinity*, A. K. Peters, 2010.
- [27] H. Rogers, Jr., *Theory of Recursive Functions and Effective Computability*, MIT Press, 1987. (See Chapter 11, especially Sections 11.7, 11.8 and the exercises for these two sections.)

- [28] A. R. Meyer, D. M. Ritchie, "The complexity of loop programs," *Proceedings ACM National Meeting, 1967*, pp. 465–469.
- [29] C. Calude, *Theories of Computational Complexity*, North-Holland, 1988. (See Chapters 1, 5.)