Digital Replicators Emerge from a Self-Organizing Prebiotic World

B. Greenbaum\textsuperscript{1} and A. N. Pargellis\textsuperscript{2}

\textsuperscript{1}Tisch Cancer Institute, Icahn School of Medicine at Mount Sinai, New York, NY 10029
\textsuperscript{2}Bonsall, CA 92003
apargellis@yahoo.com

Abstract
Some artificial chemistries model the synthesis, evolution and complexity of digital life consisting of sequences of computer operations ("opcodes") that are driven by point mutations to compete for memory and CPU time. One of us previously built Amoeba, a computer world inspired by Tierra and designed to study the emergence of self-replicating sequences of opcodes from a prebiotic world initially populated by randomly selected opcodes. Eventually an "ancestral opcode sequence" would emerge. The current version of Amoeba uses a computationally universal opcode basis set and the same addressing as Tierra. It was previously thought such changes would preclude the emergence of self-replicators. Instead, these modifications radically affect the emergence of self-replicators from the primordial soup; Amoeba exhibits a self-organization phase, after which self-replicators emerge. First, the opcode basis set becomes biased. Second, short opcode "building blocks" are propagated throughout memory space. When sufficiently dense, these prebiotic sequences combine to form self-replicators.

Introduction
Artificial computer worlds have been used to study such diverse topics as: artificial chemistry architectures (Suzuki, 2011; McMullin, 2012), the synthesis of life (Ray, 1992), emergence of life (Pargellis, 1996a), modeling life (Adami, 1995), biological complexity (Adami, 2000).

There has been considerable debate as to how self-replicators can emerge from a primordial "soup" of initially random computer operations ("opcodes"). One hypothesis is that replication may require two or more cooperating entities (Eigen, 1971; Tangen, 2010).

Amoeba is an artificial chemistry specifically designed to study the process of self-organization in a prebiotic world that eventually leads to the emergence of self-replicators. Amoeba's memory space is initially loaded with opcodes randomly selected from a set of 25 possible opcodes. The Amoeba programs compete for memory space and CPU time and evolve through point mutations (Pargellis, 2003).

The original version ("Amoeba-I") used a limited set of 16 possible opcodes and a memory topology similar to that in Avida (Adami, 1998) where virtual CPUs operated on short sequences of opcodes situated on a 2D "interaction grid" (Pargellis, 1996b). Complements of the opcodes themselves were the addresses so it was impossible to move to arbitrary positions in memory. Amoeba-I could not simulate an infinite Turing tape as there were no stacks assigned to the CPUs (Adami, 1998).

Amoeba-II added two stacks for each CPU and expanded the opcode set (Pargellis, 2001). While these opcodes formed a computationally universal set, the addressing used opcode::address pairs, making it difficult to navigate throughout memory.

Amoeba-III used a new topology for opcode memory space; the 2D interaction grid was replaced by 500 parallel "mini Turing tapes" or bands, each consisting of thousands of opcodes (Pargellis, 2003). Amoeba-III still used opcode::address pairing.

Although some biasing of the original opcode basis set was observed (Amoeba-II and –III), there was no additional self-organization of opcode sequences into "building blocks." Instead, a randomly generated sequence of opcodes, capable of self-replication, would "spontaneously emerge."

We report that a modified version of Amoeba ("Amoeba-IV"), with addressing that freely accesses memory, not only exhibits emergence, but does so using a far richer pathway.

Description of the Amoeba-IV System
The current version of the Amoeba system ("Amoeba-IV") uses the same toroidal 2D memory space used in Amoeba-III (Pargellis, 2003). The opcode basis set is similar but there are some significant differences that have radically changed the self-ordering of Amoeba's memory space, the emergence of ancestral replicators, and the diversity of those replicators.

There are several major improvements. NOPs are used for addressing as was done with the Tierra/Avida systems. The self-exam process for calculating a cell's size is more complex. Eight opcodes use the Avida methodology of a default operation modified by means of a following NOP.

The most significant change is the use of NOP-addressing. It was originally thought that using NOPs would require a prohibitively long time for emergence from a soup of random opcodes since a replicator would require at least five NOPs in addition to a minimum of seven opcodes required in Amoeba-III for primitive protobiotics. However, we find that Amoeba-IV exhibits emergence by self-organizing its opcodes through several phases. First, the opcode set coalesces into a reduced set. Second, primordial "building-blocks," consisting of short opcode-sequences, are propagated throughout the memory.

DOI: http://dx.doi.org/10.7551/978-0-262-33936-0-ch016
space. Finally, replicators emerge if the building-block density is sufficiently high.

This propagation of primordial building-blocks leading to emergence is the most significant observation of our current Amoeba research.

**Opcode Basis Set**

The set of opcodes used in Amoeba-IV is listed in Table 1.

<table>
<thead>
<tr>
<th>No.</th>
<th>Opcode Abbrev.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NOP1</td>
<td>Address label. Complement is “2”.</td>
</tr>
<tr>
<td>2</td>
<td>NOP2</td>
<td>Address label. Complement is “1”.</td>
</tr>
<tr>
<td>3</td>
<td>MALL</td>
<td>Allocate a virtual CPU for child.</td>
</tr>
<tr>
<td>4</td>
<td>COPY</td>
<td>Copy one opcode from parent to child.</td>
</tr>
<tr>
<td>5</td>
<td>DIVD</td>
<td>Initiate child CPU.</td>
</tr>
<tr>
<td>6</td>
<td>IFAG</td>
<td>If AX &gt; BX, skip next opcode.</td>
</tr>
<tr>
<td>7</td>
<td>IFAL</td>
<td>If AX ≤ BX, skip next opcode.</td>
</tr>
<tr>
<td>8</td>
<td>JMPB</td>
<td>Jump IP back to NOP complement.</td>
</tr>
<tr>
<td>9</td>
<td>JMF</td>
<td>Jump IP forward to NOP complement.</td>
</tr>
<tr>
<td>10</td>
<td>CALL</td>
<td>CALL address (similar to JMF).</td>
</tr>
<tr>
<td>11</td>
<td>RETN</td>
<td>Return to opcode after CALL.</td>
</tr>
<tr>
<td>12</td>
<td>ADRB</td>
<td>Search back for address, put in EX.</td>
</tr>
<tr>
<td>13</td>
<td>ADRF</td>
<td>Search forward for address, put in EX.</td>
</tr>
<tr>
<td>14</td>
<td>SWEF</td>
<td>Switch EX value with FX value.</td>
</tr>
<tr>
<td>15</td>
<td>TOGS</td>
<td>Toggle active stack (Stk-a and Stk-b).</td>
</tr>
<tr>
<td>16</td>
<td>PSHA</td>
<td>Push AX (BX) value onto active stack.</td>
</tr>
<tr>
<td>17</td>
<td>POPA</td>
<td>Pop active stack value to AX (BX).</td>
</tr>
<tr>
<td>18</td>
<td>ADDA</td>
<td>AX = AX + BX (BX = AX + BX)</td>
</tr>
<tr>
<td>19</td>
<td>SUBB</td>
<td>BX = BX − CX + 1 (AX = AX − CX)</td>
</tr>
<tr>
<td>20</td>
<td>BEQA</td>
<td>BX = AX (AX = BX)</td>
</tr>
<tr>
<td>21</td>
<td>AEQZ</td>
<td>AX (BX) = 0</td>
</tr>
<tr>
<td>22</td>
<td>INCA</td>
<td>Increment AX (BX) register by one.</td>
</tr>
<tr>
<td>23</td>
<td>DECA</td>
<td>Decrement AX (BX) register by one.</td>
</tr>
<tr>
<td>24</td>
<td>INCD</td>
<td>Increment DX register by one.</td>
</tr>
<tr>
<td>25</td>
<td>DECD</td>
<td>Decrement DX register by one.</td>
</tr>
</tbody>
</table>

Table 1: Description of opcodes used in Amoeba.

**Differences in the Opcode Definitions.** Ideally, an opcode basis set is computationally universal, enabling a system to develop algorithms of arbitrary complexity. But, the choice of basis set is not the only criterion for a Turing machine. Another requirement is the ability to move to any location in memory. Maley showed that the Tierra system could simulate a Turing machine, although somewhat inefficiently (Maley, 1994). Amoeba-IV addresses this issue by means of the Tierra-like NOP-addressing.

The JMPB, JMF, and CALL opcodes move the pointer to the address complement of the following NOP(s). In the absence of such a NOP, the IP will jump to the address stored in the EX address register if an opcode such as ADRB or ADRF had previously loaded the EX register.

Several opcodes (PSHA, POPA, ADDA, SUBB, BEQA, AEQZ, INCA, DECA) invoke the Avida methodology. A default operation is modified if the following opcode is a NOP. The alternate action is shown in parentheses in Table 1.

**Memory Space and Virtual CPUs**

Amoeba-IV uses a 2D memory space with periodic boundary conditions, organized into 500 “Tierra-like” bands, each with 2399 opcodes, for a total of ~1.2x10^6 opcodes. The basis set has 25 unique opcodes. A cell’s sequence is confined to a particular band. However, cells can allocate memory for their children in adjacent bands and at other start positions within those bands. We chose 2399 opcodes per band because 2399 is a prime number. This prevents cells from being able to generate an integral number of children along a band, creating a “barrier” to altering cell sizes in future generations.

There are 2000 virtual CPUs, allocated in two ways. First, at the start of a new generation, 5% of the CPUs are assigned to sequences of randomly generated opcodes. Second, a cell (opcode sequence) allocates the next CPU in the queue (using the MALL opcode) prior to copying opcodes to its child. A new generation starts when all CPUs have been allocated.

Each CPU has four numerical registers, two address registers, two stacks, and an instruction pointer (IP) that operates on opcodes. Additional parameters include the cell’s size and its IP location (band and position within the band). CPUs are accessed sequentially. Each CPU is given a slice of CPU time that is proportional to its cell’s size. Slices range from a minimum of 6 to a maximum of 100 operations.

Figure 1 is a snapshot of part of Amoeba’s memory space (color-coded opcodes) along with a schematic of a virtual CPU. An opcode sequence in one of the bands is expanded.

Figure 1: A small portion of the Amoeba memory space is shown at the top. Opcodes are color-coded; the color key is at the bottom.

In Figure 1, IPs move from left to right along a band unless jumped to a NOP-address along the band by means of a JMPB, JMF, CALL, or RETN opcode.
**Evolution and Mutations**

Replicators evolve in three ways: opcode mutations; randomly generated sequences; and cells overwriting each other’s code. Opcodes are mutated at two different times. Each time a cell copies an opcode to its child (COPY opcode) the probability that opcode is substituted by another is 0.005. Each time a cell initiates its child (DIVD opcode), the child’s opcode sequence can undergo one of three types of mutations, for a total rate of 0.10. Empirically chosen mutation rates are insertion (0.02), deletion (0.02), and substitution (0.06). Divide mutation rates above 0.20 randomize opcodes faster than building-blocks critical for replications are propagated while rates below 0.05 overly bias the opcode basis set; NOPs dominate and some opcodes critical for replication become too infrequent.

At the start of each new generation, 100 sequences, each consisting of one, two, or three randomly selected opcodes, are randomly distributed throughout memory space. The probability of inserting a random sequence is peaked at the middle of the 500 bands. The middle bands tend to be “melted”, but the edge bands have no random sequences.

There is no write-protection in Amoeba. The MALL opcode only defines the start position for a child’s opcodes and assigns it a virtual CPU.

**Anatomy of a Typical Self-replicator**

Figure 2 shows the anatomy of a typical self-replicator.

![Figure 2: Anatomy of a replicator. Run: 06-08-6140. Gen: 3.499M.](image)

The opcode color-coding is as in Figure 1. This replicator has four “replicator genes” (shown on the left-hand side): Self-Exam (ADRB, NOP1, ADRF, NOP2, SUBB), Copy Loop (NOP1, COPY, INCA, IFAG, JMPB), Biological where a virtual CPU is allocated and then initiated (MALL, DIVD), and Reset registers (INCD, POPA, JMPB). Note that the genes overlap and/or can be split into multiple pieces.

The main feature of the Amoeba-IV system is the self-organization of the initially random distribution of opcodes in memory.

**Self-organization Leads to Emergence**

The development and evolution of an Amoeba system from the initial random distribution of opcodes to the emergence of “ancestral self-replicators” occurs during three main stages: prebiotic, protobiotic, biotic (Pargellis, 1996a). The prebiotic phase coalesces the original set (alphabet) of 25 opcodes into a reduced, biased set of about 15 to 20 opcodes. This reduced set continues to self-organize, generating short sequences of n-opcodes (“n-ops”) that propagate critical building blocks required for replication. An inefficient proto-replicator emerges, usually within a million generations, when the building-block density is sufficiently high. Mutations drive the proto-replicators to evolve into robust replicators that eliminate unneeded opcodes and unroll the copy-loop (multiple {COPY, INCA} sequences per loop).

Figure 3 shows the distribution of (binned) emergence times for a set of 25 runs out of a total of 49 runs.

![Figure 3: Distribution of self-replicator emergence times.](image)

The opcode basis set always biases into a consolidated set of about 20 opcodes within the first 50,000 generations. Nearly half of the emergences occur within the first million generations. One cause for the steady drop in emergence probability with time is the continued biasing of the opcode basis set. Significantly, after ten million generations more than 25% of the opcodes are either NOP1 or NOP2. This reduces the frequencies of other opcodes critical for replication, including conditional opcodes (IFAL and IFAG) and branching opcodes (JMPB and JMPF).
**Self-organization**

The propagation of n-ops throughout memory can be visualized in the (partial) screen-shots shown in Figure 4. (A) Initial random opcode distribution, (B) prebiotic self-organization, (C) post-emergence dominated by self-replicators.

![Figure 4](image)

The initially random opcode distribution of Figure 4A becomes partially ordered during the self-organization phase of Figure 4B. Some opcodes are replicated sequentially, shown by the horizontal lines of a single color. A series of snippets are visible in the left-hand third of Figure 4B. Figure 4C shows that after emergence, the memory map consists of thousands of replicators across bands and within bands.

We can quantify the memory-map organization of Figure 4 by calculating the two-position opcode correlations within a band. Opcodes are uncorrelated at the beginning of a new run. Once self-replicators emerge, opcodes are correlated over multiples of the replicator lengths. We have calculated the correlations within bands over windows 100 opcodes wide. We do not show those results here because the correlations do not add significantly to what we already learn from visual inspection of the memory space.

Size Distribution of Opcode Sequences. The self-organization and increasing frequencies for selected n-ops leads to a population of children with steadily growing sizes. Figure 5 shows the growth in the sizes of children. Initially, the sizes are mostly one to three opcodes as this is the size range for the (100) randomly generated sequences placed throughout the Amoeba world at the beginning of each new generation. Longer opcode sequences become more prevalent, eventually leading to the emergence of a proto-replicator.

Frequency Growth of Critical n-opcode sequences. A fundamental difference between Amoeba-IV and earlier versions is that the “ancestral” protobiotic does not suddenly emerge through some fortuitous combination of opcodes. The inclusion of NOPs for addressing in the Amoeba-IV state machine makes the probability of such an ancestor’s spontaneous emergence highly improbable. To date, the smallest self-replicators contain about 15 opcodes (14 if the replicator “guesses” its size, e.g. does not have the self-exam gene). This results in $2^{14} \approx 10^{19.6}$ combinations of which a tiny (not easy to estimate) fraction are self-replicators.

Surprisingly, rather than losing self-replicator emergence, the Amoeba memory space self-organizes over a period of several hundred-thousand generations by propagating opcode sequences of ever growing length and complexity. Eventually, in about half the runs, a proto-replicator ancestor emerges that quickly evolves into a population of robust replicators.

**Case Example: Self-organization to Emergence**

In this section, we analyze one example for the emergence from an initial primordial “soup” of random opcodes. We first present the anatomy of the emerged replicator, followed by data on the self-organization that leads to the propagation of building blocks for the “replicator genes.”

The most complicated replicator gene is the copy-loop because it includes machinery for copying opcodes from the parent to its child, a branch opcode to repeat the loop, and a conditional check that breaks out of the loop. This means a typical copy-loop consists of some version of [NOP, COPY, INCA, IFAG, JMPB]. As Figure 2 above shows, this can be complicated in cases where parts of other genes, such as the Self-Exam gene, are embedded in the Copy-Loop gene.

In this section, we choose a somewhat unusual case using a CALL-RETN copy-loop rather than the more common loop
ending with a JMPB opcode. We chose the CALL-RETN case because there are no extraneous opcodes (parts of other replicator genes) embedded in the copy-loop, despite the fact that the CALL-RETN loop has six opcodes instead of five. (Actually, the example shown here unrolled the {COPY, INCA} combination very quickly. We will show the building-block development that led to this.)

**Anatomy for a “CALL-RETN” Replicator.** The anatomy of a “CALL-RETN” replicator is shown in Figure 6. There are 14 unique opcodes in this replicator that are useful for replication. Irrelevant opcodes (introns) have been neglected for brevity. The CALL-RETN replication method is rare because it requires two opcodes (CALL and RETN in that order) for closing the copy-loop. Most replicators just use one opcode (JMPB) to close their copy and reset loops.

The CALL opcode is handled similarly to the JMPF opcode; CALL first looks for one or more NOPs immediately following. In the absence of subsequent NOPs (as in this example), CALL will check its EX-register for any loaded address and then jump to that address. In this case, the ADRB-NOP2 combination has loaded the EX-register with the complementary address, “1” (NOP1). The RETN is used both for the COPY loop and resetting the IP after the DIVD. (Amoeba ignores the DIVD operation unless the parent had first allocated a CPU for its child by means of the MALL opcode). The ADRB is replaced with ADRF in only 2000 generations. This is frequently seen in Amoeba as a parent automatically puts the beginning of its child sequence in the child’s CX register. The only advantage of retaining the ADRB opcode is to capture “rogue IPs”, enabling the host to commander the virtual CPUs associated with other cells in the same band.

**Growing Single-Opcode Frequencies.** Initially, all 25 possible opcodes are equally distributed and the frequency (fraction, \( f(m_j) \), of all 1.2 million opcodes) is the same; \( f(O) = 0.040 \) for all \( O_j \). However, the frequencies for some opcodes useful to propagating sequences of opcodes preferentially grow at the expense of other opcodes.

Figure 7 shows the increase in frequency over time (left-hand scale) for selected single opcodes (1-ops). Emergence occurred at about 292,000 generations (labeled, vertical line). On the right-hand scale in Figure 7, we plot the “monomeric opcode entropy”, \( S(O) \), as a quick check into the status of self-organization of single opcodes and given by,

\[
S(O) = -\sum_{j=1}^{D} \frac{m_j}{M} \ln \left( \frac{m_j}{M} \right). \tag{Eq. 1}
\]

where \( D = 25 \) is the size of the alphabet (number of unique opcodes in the basis set), \( M \equiv 1.2 \times 10^6 \) is the size of the memory space (total number of opcodes), \( m_j \) is the number of occurrences (counts) for the \( j \)th opcode, given by the symbol, \( O_j \), and \( \ln \) is the natural log. An estimate of the effective size of the opcode basis set is \( E(O) = \exp(S) \). For the initial, equally distributed opcodes, \( S(O) = 3.218 \) and \( E(O) = 25 \). By 500,000 generations, the entropy drops to \( S(O) = 2.935 \), indicating the useful part of the basis set shrinks to about 19 opcodes. Most runs evolve a basis set of 15 to 16 opcodes.

---

**Figure 6**: Anatomy of a replicator that uses the CALL/RETN combination for both the (extended) COPY loop and IP return. Run: 11-10-2015_25271. Gen: 321,982.

**Figure 7**: Growth of 1-ops and commensurate decrease in entropy for the CALL-RETN replicator of Figure 6. Run: 11-10-2015_25271.

Initially, 1-ops conducive to replication grow in frequency: NOP1, NOP2, MALL. The NOP1 and NOP2 frequencies begin to increase about 70,000 generations before emergence. The NOPs are arguments for the CALL and ADRB opcodes.

The ADRB opcode is not required in order for a cell to calculate its size and load that value into its BX-register. Amoeba automatically loads into a child’s beginning location

DOI: http://dx.doi.org/10.7551/978-0-262-33936-0-ch016
in the band into the CX-register when a parent first allocates memory (using the MALL opcode) for its child. However, ADRB is useful for resetting the CX registers of rogue IPs, essentially hijacking other virtual CPUs for the host. Once the replicator population becomes dominant, the ADRB is no longer useful. In this case example, the ADRB opcode mutated into ADRF which also loaded the EX-register with the NOP1 address (used by CALL). Consequently, f(ADRFR) doubles and f(ADRB) drops.

After emergence, COPY and INCA become more prevalent when the copy-loop is “unrolled” and the combination {COPY, INCA} is repeated.

Growing Multi Opcode Frequencies. The co-occurrence of two opcodes, and therefore the degree of opcode self-organization, can be quantified using the mutual information (MI) measure (Manning, 2000),

\[ MI = \sum_{(O_i, O_j)} p(O_i, O_j) \ln \left( \frac{p(O_i, O_j)}{p(O_i)p(O_j)} \right). \]  
(Eq. 2)

where the sum is over all \( 25^2 = 625 \) possible opcode pairs for our basis set of 25 opcodes. The argument in the sum is the pointwise mutual information, \( MI(O_i, O_j) \), which is the log of the odds-ratio of opcode pairs and is zero in the absence of any correlation.

We plot \( MI \) versus time with the dashed blue line in Figure 8 (right-hand scale). The MI shows clearly that self-organization has occurred. Critically, what we previously noted as the time of emergence coincides with the maximum increase in the mutual information. Scientifically, this is a key finding. We demonstrate the degree of ordering for the joint probability versus the uncorrelated single probabilities for a select set of opcode pairs (2-ops) by plotting the time evolution of the “odds-ratio”, the argument of the natural logarithm in (Eq. 2), in Figure 8. We used the CMU toolkit for counting n-ops (Clarkson, 1997).

A key observation is that 2-ops critical to development of the CALL-RETN loop, {CALL, COPY, INCA, RETN}, grow in abundance at least 50,000 generations before emergence during the self-organization period. The ADRB opcode gets replaced by the ADRF opcode once it is no longer useful after emergence and the {RETN, NOP1} drops after emergence because introns are inserted between the RETN and NOP1.

Development of the Copy-Loop Building Block. The copy-loop building block is noteworthy since the primordial COPY-RETN loop, {CALL, COPY, INCA, RETN}, is clearly a prebiotic building-block; it is impossible for an IP to break out of this loop!

This sequence copies opcodes throughout memory, but without a conditional check (IFAG or IFAL) the IP can never break the loop and initiate a child. Nevertheless, this propagator is a useful building block, capable of propagating itself and other opcodes. An insertion mutation could have modified this building block prior to emergence by inserting an IFAG conditional check before the RETN opcode.

Figure 9 is a schematic showing how the copy-loop is built from the CALL-RETN building-block.

The CALL opcode calls the NOP1 address (see Figure 6). Usually, the CALL opcode is followed by a NOP and the call is to that NOP’s complement. In the absence of a subsequent NOP, the CALL looks into its address register, EX, to see if it has an address. In this run, the previous ADRB opcode loaded the EX-register with NOP1, the complement to NOP2. Note: the {ADRB, NOP2} combination was many opcodes prior to the CALL-RETN loop in prebiotic propagator sequences. There was also a subsequent NOP1 in the primordial form of the CALL-RETN loop. An intron preceded the NOP1 during pre-emergence.

The vertical dashed blue lines show the time range over which each of the sequences exist. The solid blue diamonds show the time range over which each of the sequences exist. The solid blue diamonds are when we first saw the sequence in either a log-file or a memory snapshot. Times of extinction are shown by solid blue circles. Once a sequence occurs, it is propagated for
many generations until replaced by a more viable alternative. For example, the \{CALL, COPY, INCA, RETN\} sequence persists until the copy-loop is unrolled (generation 239,000).

**Emergence of Ancestral Replicators**

A consequence of the openness of the Amoeba-IV system is the diversity of ancestral replicators that emerge from the self-organizing, initially random “soup” of opcodes.

It is a challenge to identify the emergence of an “ancestor” in Amoeba-IV. Ancestors are logged in a file when a parent has faithfully copied itself to children for at least four generations. However, the early “protobiotic” replicators do not copy themselves with any fidelity. It typically takes several thousand generations before a “faithful” replicator is generated and logged. During this time, millions of cells have been initiated and some small subset of that number will eventually lead to an ancestor that faithfully copies its opcodes to its children. It is possible that an ancestor emerges out of a hypercycle of interacting components (Eigen, 1971) but this type of interaction is nontrivial to track. One can examine the “world snapshots” that are periodically saved, but each snapshot is a list of the entire Amoeba memory; the analysis of any interactions within that map is difficult.

The next examples show the rich diversity of replicators that is one of the novel outcomes of the modified Amoeba-IV artificial chemistry.

**“Size-guesser” Protobiotic.** An example of a class of robust proto-replicator that has emerged in several runs is shown in Figure 10. This is an inefficient replicator; it does not include the “self-exam” gene so it “guesses its size” and cannot generate a complete copy of the parent until the 3rd child.

About 200,000 generations later, this proto-replicator evolved into a robust replicator that used ADRF and SUBB opcodes to properly calculate its size.

**“Conditional Ladder” Replicator.** The anatomy of a “conditional ladder” replicator is shown in Figure 11.

![Figure 10: Anatomy of a replicator that guesses its size; the self-exam gene is missing. Run: 06-17-2015_54809. Gen: 1.042M.](media/image)

This replicator’s COPY loop is inefficient because it includes the entire cell’s opcode sequence. An “IFAL-ladder” is used to avoid prematurely resetting the cell’s registers while copying its code to a child. When done copying, the replicator’s 4th IFAL check fails and the BX register is set by means of the 2nd SUBB opcode. The IFAL ladder checks then “fail” and the replicator initiates its child (DIVD) and resets its AX register (POPA).

**Discussion and Future Research**

There are several other interesting observations in addition to the success in generating self-replicators due to opcode self-organization from a primordial soup consisting of Tierra-like opcodes and addresses.

Many proto-replicators lack some of the replicator genes shown above in Figure 2. An example of a proto-replicator without the self-exam gene was shown in Figure 10. Other replicators do not retain their IP. This was observed in earlier versions of Amoeba where cells are members of a colony. The

DOI: http://dx.doi.org/10.7551/978-0-262-33936-0-ch016
colony members lose their IP after generating a single child. Their IPs then execute opcodes “belonging to” other cells further down their respective Tierra-band.

We observe runs where replicators cheat and get a larger CPU time-slice. This issue arises because the time-slice is proportional to a cell’s size. Earlier versions of Amoeba automatically incremented the AX-register as part of the COPY opcode; the size of a cell was the number of opcodes copied. Amoeba-IV now requires the INCA opcode to increment the AX-register. This raises an interesting question: how to determine the size of a child when the DIVD opcode initiates it? We cannot increment the size every time a COPY is used since prebiotics can copy the same opcode many times (they lack the INCA opcode). Currently, we use the AX-register value to determine the size. The “cheaters” copy their replicator sequence, typically about 30 opcodes. Then, prior to the DIVD opcode, they use multiple ADDA opcodes to increase the AX-value to many times the cell’s size. We are still investigating how to prevent this parasitic behavior without decreasing replicator genome diversity.

The Amoeba systems have always been dominated by variants of a single species after emergence. We have never observed two radically different species co-existing. Viruses occur, but are quickly eliminated when the host mutate. For example, the retention of the ADRB opcode, even though not required for a self-exam, effectively “highjacks” viral IPs and their associated CPUs. We believe a variation in the externally imposed fitness landscape may enable alternative species, as well as parasites and hosts, to co-exist for extended times. Amoeba-IV partially addresses this by varying the rate at which random sequences are distributed throughout the bands. However, replicators are still able to quickly scatter their children throughout all bands. There are several options to consider: slowing down the rate at which children are spread throughout the bands, imposing write-protection when a cell allocates memory for a child (similar to Tierra), or modifying the time-slice parameters for different regions in memory so that replicator sequences of different sizes would find some parts of memory inhospitable.

Only half (25 out of 49) of the Amoeba runs exhibit emergence (see Figure 3). We observe that the probability of a replicator emerging after about one million generations is low. This implies that there are some self-organization processes that can lower the probability of emergence. One observation is that the NOP1 and NOP2 opcodes steadily become more prevalent in runs without emergence. This reduces the likelihood of creating building-block propagators because other opcodes critical to replication occur less frequently.

The earlier Amoeba versions I and II demonstrated that the probability of a randomly generated opcode sequence being a replicator increased with size (number of opcodes). It has been argued that computation-universal chemistries such as Avida or Tierra would exhibit a probability that decreased with increasing size (Adami, 1998). Preliminary observations with Amoeba-IV indicate this may not be the case – ancestral replicators typically include 200 to 300 opcodes – and this aspect of our findings warrants further exploration. While ancestors in Amoeba-IV are generated from building-blocks that are propagated during the self-organization phase, these “genes” are generated multiple times in huge ungainly “protobiotics” with sizes ranging up to the maximum currently allowed (450 opcodes per cell).

Another topic of interest is the issue of “propagators,” “protobiotics,” and “replicators.” Amoeba-IV has shown it is very difficult to identify precisely when an ancestral replicator “emerges.” It may be that groups of propagators form hypercycles that eventually lead to robust self-replicators (Eigen, 1971; Eigen, 1981; Eriksson, 2006). We are currently investigating this by logging sequences that have been generated at least 50 times, regardless of whether or not the sequences are the same as their parental sequence.

Acknowledgements

We are grateful for fruitful conversations with Stanislas Leibler who also inspired this work. We owe a great deal to the folks at IAS (Princeton, NJ), especially Bernard Chazelle.

References