943

A Primordial Soup Environment

Jeffrey Putnam
New Mexico Institute of Mining and Technology
jefu@nmt.edu

December 28, 1992

Abstract

Stew is a system similar to Tierra, but using a rather different machine model. The intent of Stew is to enable exploration of evolution in a spatially distributed machine model.

1 Introduction

Tierra [1] has been a very successful project involving genetic evolution of individuals. Each individual is essentially a "virtual computer" that runs a basic instruction set in a large shared memory ("the soup"). Instructions can execute with errors and this leads to evolution over time. The individual "virtual computers" are considered to be surrounded by a "semi-permeable" membrane which allows other individuals to read and execute the instructions, but not to write to them.

Tierra has shown that a simple model can give rise to complex behaviors with evolution giving "parasites", individuals resistant to those "parasites" and generally more efficient self copying individuals.

Stew was inspired largely by Tierra and also by several talks at the ALIFE III workshop in Santa Fe. One of these talks (by Buss and Fontana) presented something that looked rather like Tierra but with the lambda calculus as its fundamental machine model. It prompted the question: "How fragile are different machine models?" as well as "how adaptive".

The original idea was to modify Tierra to work in a two and a half dimensional space. Individuals would occupy linearly adjacent locations as in Tierra. Each set of these memory locations would reside in a two dimensional grid and additional instructions would be added to the system to allow each individual to address memory in those adjacent locations. Thus, the system could be seen as a two dimensional grid of columns. Each column would then receive the same number of instructions on each pass through the grid.

Another change to Tierra considered was to loosen the "semi-permeable" membranes surrounding each individual and to allow another individual to write through the membrane and into the individual under certain conditions, such as when the individual wanting to write had more energy than the recipient. This would enable true parasites to form.

Upon consideration of the Tierra code and the realization that the changes needed would necessitate a very large amount of coding, it was determined that a new system would be a worthwhile endeavor as it could provide for more flexibility and make instruction set modification simpler.

The system has been constructed and has run several hundred million instruction, however, interpretation of the data has not kept pace with the results. Some preliminary results are discussed below.

2 Stew Design

The design of **Stew** had several goals. The general intention was to build a system that would model (at least a computer scientist's view of) a molecular soup. Specific goals included:

1. The ability to spatially distribute individuals and move them around
2. A separate address space for each individual
3. The ability to manipulate address spaces without using addresses
4. Granting individuals the capability to write to the "memory" of other individuals.
5. Enough flexibility to make further changes easy.
6. As in Tierra the basic building blocks are both instructions and data.
7. Rather than making instructions execute with errors, every instruction in the system is potentially mutated on a regular basis.

Two model interactions between individuals were also considered important in the design. Making a first implementation capable of these interactions was a prime goal as well as the general ideas above.

The first of these was to make the individual elements of the soup capable of doing essentially a genetic crossover. Thus, a suitably programmed individual should be able to attach itself to two other individuals and to intermix them to produce an individual that consists of some of the first and some of the second. Then with (eventual) read and write errors, this individual essentially builds a mutated and crossed version of other individuals. It should be possible for the individual to use itself as one of the contributing individuals, thus reproducing sexually.

The second interaction was to allow an individual to write into another individual when certain constraints were satisfied. This would enable the process of killing off weaker individuals to be not a matter for the programmed system, but a natural part of the system's workings.

As in Tierra, there is a way for an individual to create a "child".

The system thus conceived eventually evolved to consist of the following basic elements (the terminology is borrowed from chemistry without apology for any misconceptions) :

Atoms An atom is a single instruction or component. Currently atoms take values from 0 to 255 (thus could be represented in a single byte). Atoms are always non-negative. Atoms are organized as sequential lists or streams. Currently less than 256 instructions are used so remaining atoms are used to represent instructions more than once. Thus the `add` instruction might be instructions number 1, 51, 101, 151, 201, 251. This allows for flexibility in adding to the instruction set. Labels may be set in atom streams by using the instruction `label` followed by the atom that represents the label.

Contexts A context is an array of atoms along with a position. Contexts are used to represent instruction execution as well as the state of any reading and writing. The position thus represents the PC of the abstract machine, as well as a read/write pointer, depending on context. There are four context "registers" in each abstract machine. One of these is the execution context, one is a read context, one a write context, and the fourth may be used for a scratch "register". Each molecule also has a context stack. Jumps to atom streams of any type are possible and are accomplished by setting the execution context to be another context.

Molecules A molecule is the fundamental virtual machine in **Stew** . It consists of an array of atoms, some execution state and some information that allows for interaction with other molecules. The molecule is the central element in this system and will be described in more detail below.

Cells A cell is a location that holds molecules. Molecules can move from cell to adjacent cell randomly as long as all cells attached together move in the same way. All molecules in a cell can interact, thus a cell is considered to be a small area in space.

3 Molecules

The molecule is the fundamental virtual machine in the system. The computational aspects of the molecule were borrowed from Forth or Forth like languages. All computation is performed on an operand stack where the operands are atoms. Thus, to add two atoms, the atoms are pushed on the stack, then the `add` instruction is executed which pops them both, adds them and pushes the result back on the stack. Operations that perform computation or stack manipulation include basic arithmetic, logical combinations and some operations that rearrange elements on the stack. The instruction set is given in Appendix A.

Computation may involve errors. Each error (for example, `pop` from an empty stack or `divide` by zero) increments the damage counter in the molecule by 1. Each instruction executed (successfully or not) decrements the damage counter by some (configurable) amount. When the damage counter reaches some upper limit (which can be configured at execution start up), the molecule dies and is removed from the population. There is also a `die` instruction which immediately kills the molecule.

The state of the molecule also depends on several contexts. These includes four context in an array - the current execution context, a read context, a write context, and an unnamed context (for scratch use) and a stack of contexts which can be used as a subroutine stack or as a convenient place to hold contexts until needed. Several instructions are included to manipulate contexts and the context stack, including basic stack operations. Instructions are also provided that create contexts in different ways. For example, one way to implement a `jump` is to create a new context that copies the execution context, then to search it for a label. The result is that the (copied) context now has its pointer set to the label (if any is found). To finish the jump the new context is then set to be the execution context.

Molecules exist in a world (consisting of a set of cells) with a certain geometry. Currently this world is one dimensional world and the ends wrap. The size of this world may be set at run time. Molecules move randomly from one cell to adjacent cells where the probability of movement from one cell to another depends on their relative number of cells. Thus cells with many occupants tend to lose their population to adjacent cells.

In order to interact with other molecules each molecule has four "arms". The first arm is always attached to (the molecule) itself. The other three can be used to grab other molecules so that the molecule can read or write them. This works as follows: Each molecule contains a "species I. D." (a sequence of four bytes) which may be used to identify it. This is inherited from the molecule's parent (perhaps with mutations). A molecule can load a "pattern" from its operand stack and `wait` for a molecule whose id matches the pattern. A molecule matching the pattern is attached to one of the arms and a context for that molecule is created and put on the top of the context stack. Zeros in the pattern are wild cards and will match anything.

Other important instructions are `detach` which frees a molecule from the arms and `probe` . In order to write to a molecule, an attaching molecule must perform the `probe` instruction. `probe` compares the damage of the probing molecule and the probed molecule and randomly determines if the prober can write. This is done by picking a random number r_1 in the interval $[0, damage]$ for molecule 1 and one r_2 in the interval $[0, damage]$ for molecule 2. If $r_1 < r_2$ the probing molecule gains write permission. Once write permission is granted it will not be lost until the probed molecule is released by the probing molecule. `probe` also increments the damage in both molecules by some random amount. This was intended to discourage molecules from simply sitting in a tight loop and probing others.

4 How the World Works

When started the system usually reads a parameter description file. This may set such system level parameters as the number of molecules to begin the system, the number of "generations", the mutation level and so on. It is also possible to set the first generation to s specific mixture of molecules, so lines may be entered that describe the number of molecules to make of a specific type, their "species" and the file that contains the code.

The world is created and seeded with these first molecules. Then the specified number of generations is executed. At each generation, the set of molecules is examined and any molecule not waiting is given the chance to execute some number of instructions (currently 10). The execution context of that molecule is used to find the sequence of atoms (instructions) to execute. Instructions are then executed until either the total number of instructions in the generation is reached, until the molecule executes a `wait` instruction or until the molecule damage reaches the death threshold.

When a molecule dies it releases any molecules it may have attached as well as any children it may be in the process of constructing. The children may be incomplete, but they are marked as alive and set loose just as if they had been born normally.

All molecules still alive and not waiting are thus given a chance to run. This happens in the order in which they are found in a global set of molecules. Since this set is a hash table, this order is more or less random.

All molecules in this global set are then mutated. Each atom in all molecules has a chance of a bit changing (the probability of this may be set in the configuration file).

All the new molecules (those born during the course of the generation) are then added to the global set and to the same cell as their parent. All molecules that died during the generation are now scavenged. A message is printed for each molecule (either newborn or dead) giving their makeup and status. We must print out the entire makeup of the newly dead molecules as the mutations applied over their lifetime may change their structure.

The molecules are moved from cell to cell randomly with "population pressure" the primary determination for the way the molecules move.

Finally, each cell is checked with each waiting molecule to find matches. If a match is found for a waiting molecule, the matching molecule is placed on an empty "arm" and the waiting molecule is awakened so that it may run in the next cycle.

The number of live molecules is compared with the "goal" population and a new death threshold is computed. This threshold is used to determine if molecules are dead on the next generation. The threshold can never be reduced below one half the starting threshold.

On each generation the number of molecules still alive is printed to give a way to follow the growth of the population.

5 First Molecules

It was initially intended that the system would have no external method to kill off molecules (except by the natural death mechanism above), so an early molecule constructed was the Reaper. This molecule sets up a wild card pattern so that it will match any molecule in the same cell as itself and waits for the system to find a match (the exact instructions in this function are presented in Appendix B). When a match is found, the molecule is attached and the `die` instruction is inserted at the head of the instruction stream. In this way, when the molecule comes to the end of its instruction stream and returns to the beginning the `die` instruction will kill it off. Since molecules with more damage are more prone to give write permission (as above) this tends to kill off the weaker (more damaged) molecules and leave the live ones.

In order to generate new molecules a "Cloner" was also written. The cloner waits for any molecule (with the same wild card pattern as the reaper) and then copies the molecule into a new molecule and detaches both the new and old molecules. Cloning takes rather more time than reaping (reaping needs only insert one instruction into the target molecule which takes on the order of 5 instructions, cloning takes about 10 instructions for each atom in the target).

The first experiment with the system consisted of running it with some number of clone molecules and some number of reapers initially. It proved difficult to adjust these numbers to produce a stable population, either the population grew exponentially or decreased to zero. It also developed that with anything but a zero mutation rate the reaper eventually mutated sufficiently that it could not maintain the population at anything near stability. Thus the program was modified so that the threshold for death would be reduced as the total population increased. This threshold (by default 100) would decrease asymptotically to zero as the population increased to infinity. However in even the most populous experiments, the death threshold never went below about 80. Thereafter, populations were controlled by this mechanism rather than by the use of the reaper molecule.

Generally, populations showed an initial very slow growth, usually with new molecules produced by molecules executing instructions that had mutated - either resulting in their death or in releasing any partially formed child molecules. After 32 generations (enough to reproduce a cloner), a sharp jump is observed when all the children produced by the cloner are released from the parent molecules. Because of mutations, the growth after that point is more even but with another large jump at about 64 generations where the third generation of molecules is born.

As the molecules accumulate mutations the less successful ones die and if the mutation rate is high enough the death rate soon overtakes the birth rate and the population begins to shrink. After a while the population most often reaches a stable point. However, this stability is not caused by the death and birth rates reaching the same point, but usually because the final molecules have all executed the **wait** instruction and are waiting for patterns that no longer exist in the population. Since molecules do not change their damage status during waits, a waiting molecule that never finds its pattern may sit in a wait essentially forever. It is not clear how to resolve this problem and keep molecules alive and active.

Another experiment involved building two species of molecules. The

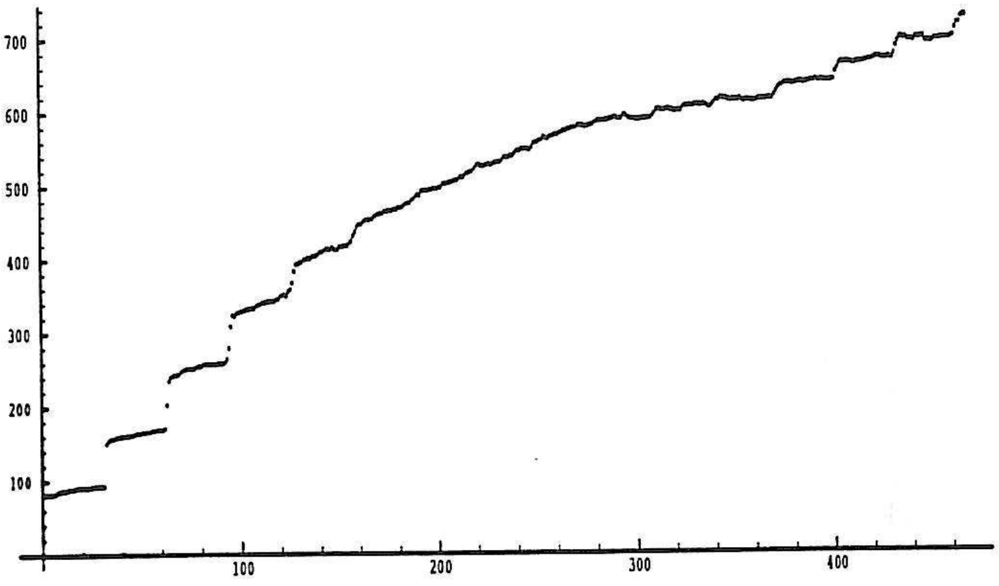

Figure 1: Population VS Time

first cloned members of the second and vice versa. Until the cutoff for small molecules (see below) was implemented, the population grew rapidly (over the course of several doublings which occurred every 32 generations) and rapidly slowed the simulation down unacceptably. With the cutoff for small molecules, a run of almost 500 generations was made and some interesting properties emerged.

The graph in 5 shows population versus time for this problem. Note the discontinuities at 32 generation intervals between generations 0 and about 160. These discontinuities blur then essentially disappear for about 200 generations then reappear about generation 280. It is not clear from the output of the program just why this is occurring. In large part this is due to the difficulty of understanding the programs produced by mutation over time.

5 and 5 show size distribution for individual molecules. In 5 the sizes are shown for the entire run. 5 shows the sizes for only those molecules of size less than 50 atoms. The large molecules in 5 seem to have been born from more or less normally small molecules. One, with a size of 455 seems

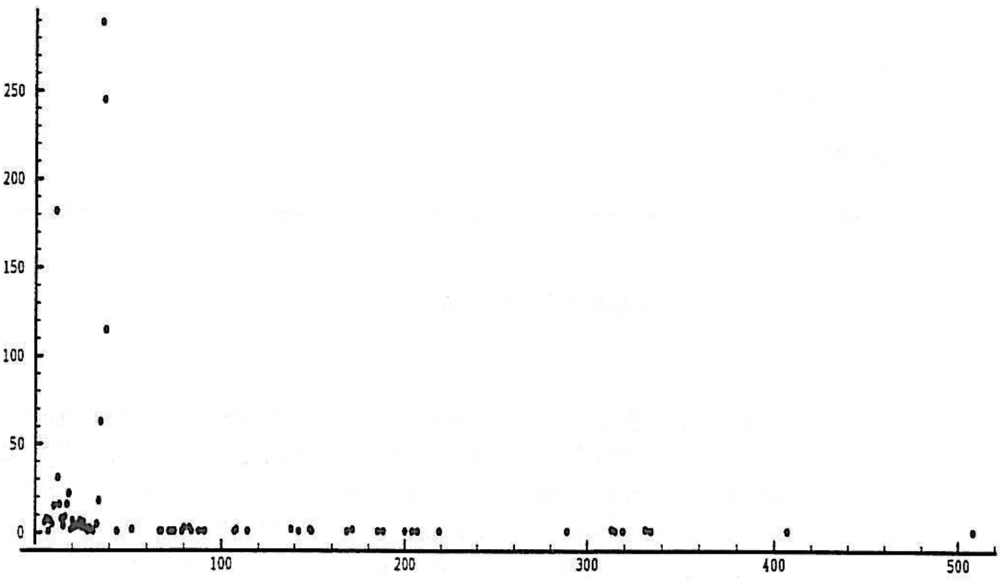

Figure 2: Individual sizes

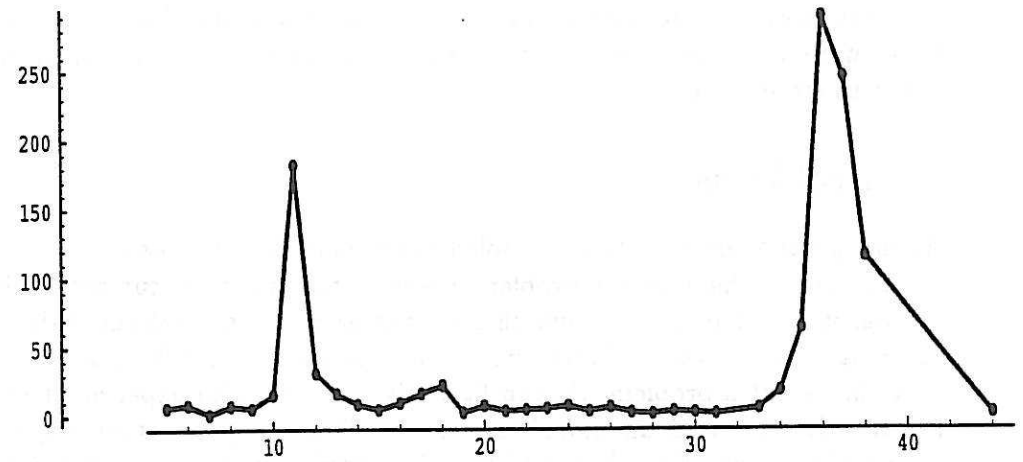

Figure 3: Individual sizes (up to size 50)

to be mostly made up of copies of mutated versions of the original copier. Its parent is a molecule of size 34 which when born builds a pattern and waits for it, then executes what seems to be a nonsense pattern. Since it had not died when the output was examined it was impossible to tell what it had mutated into. Another long molecule consisted mostly of repeated copies of the `literal` instruction which just copies the next atom onto the operand stack.

5 shows the distribution for molecules less than 50 atoms long. The higher peak (between 36 and 38 atoms) consists of copies of the original copier molecules and slight mutations from those. The 36 atom peak comes from a number of very slight mutations of the original 38 atom cloners, but for the most part, these molecules seem to attach molecules then just read from them, no writes are performed. For these to have reached such a peak in the population they must be created from simple mutations of cloners and be relatively stable.

Another peak at 11 atoms consists of a large number of atoms that first appears near generation 100. This simply seems to be a remarkably successful (there are many variants with only trivial mutations) simple loop that does nothing. Several very similar variants come from molecules that seem to be copying part of their own atom stream into a child molecule.

Thus, it seems that from a kind of sexual reproduction where molecules of different "species" interact to copy each other a set of molecules (which seem to be more or less generated independently) develops that copy themselves, albeit to sterile children.

6 Problems

At this point there are several problems that must be addressed.

The first is the deadlock problem where all the molecules currently alive are waiting for molecules to match their patterns, but no molecule existing can match that pattern. Certainly, one way to address this is to determine that this is not a problem, that in fact this is a successful experiment that has merely produced an unfruitful result. There may be other ways to resolve this as well. One that will be implemented shortly is to allow pattern matching to be more fuzzy and to increase the level of fuzz allowed as the molecule waits longer. That is, instead of a pattern match returning either true or false, let it return a number between 0 (strictly false) and 1 (strictly true). When the molecule first waits, it will only find matches where the pattern matcher returns 1. Over time the level at which the pattern matches will be reduced until it eventually reaches 0 at which time any pattern will match. This might also eventually allow the molecule to specify a pattern match level.

Another way to attack the deadlock problem is to allow pattern matching to be subject to errors.

Another serious problem at this time is the run time required by these experiments. It is not atypical to have run times upward of several days for large problems (say over 300 molecules) and the program renders the machine it is running on almost unusable by anyone else. In our local computer environment, this is unacceptable since our resources are quite limited. Another change contemplated will run a simulation over a number of machines, with each machine running a single cell. With pauses for end of generation synchronization and communication, this should make the perceived impact on machines much less.

Molecules may also be trivial. Thus a molecule that consists only of a nop can do nothing and takes up time and space without accumulating damage. Such a molecule will run forever, or until mutation changes the instruction. This raises the population and the run time without adding any significantly interesting behavior to the population. One solution (currently implemented) does not add any sufficiently small molecules to the population. The cutoff size is 5 by default, but may be configured at start up time. This leads to much smaller populations and therefore to faster runs. The ability to configure this to change over simulation time might allow even faster runs and more selective culling of unfertile children (such as the 11 atom child discussed above).

There remain several coding errors in the instruction set. This does not seem to be a major problem in the simulations as the instruction set is more or less arbitrary and thus the molecules develop according to what the instruction set actually means, but it does complicate interpretation.

7 Conclusions and Further Work

Stew, an intellectual descendant of Tierra, modifies and extends the Tierra model of computation by introducing a means for one individual to write into another, a topology for the interactions of individuals and for their motion, and a simpler model for mutations and errors in instruction execution.

While millions of instructions have been executed, few of the experiments have resulted in more than fruitless stasis as yet. There are several problems that require solution before more extensive experiments can be run.

There are a number of further extensions to this system that will be implemented to enable better control of the system and to perform different types of experiments.

For example, it might be interesting to set up the underlying array of cells in such a way as to have higher mutation rates in one cell than in the rest. This would mean that more of the molecules would remain stable while the population would have a continuing source of mutated individuals.

Since the instruction set is rather sparsely occupied at the moment, it is relatively easy to add instructions. One that might be of interest would be to add movement to the molecules (perhaps undirectional). Molecules with high movement factors would change locations relatively quickly and would thus be more likely to be grabbed by other molecules and copied.

The system currently does not implement dumping of the configurations of molecules which means that it is not possible at this point to look and see if clusters of molecules might form in a kind of symbiotic relationship. This might, given the way the simulation works, actually appear relatively early as grabbing molecules is easy.

Another very interesting potential change (in fact, the change I initially wanted to make to Tierra) would be to distribute instructions in each generation uniformly to cells, rather than to individual molecules. This might produce true symbiosis and parasitism.

A Instruction Set

Instruction	**comment**
`literal`	the next atom in the instruction stream is pushed on the operand stack
`label`	this and the following atom are ignored during instruction
`ouch`	causes small damage to a molecule
`pop`	pop the top of the operand stack and discard
`times`	multiply the two numbers on the operand stack and push the result
`minus`	subtract (as in times)
`add`	addition
`divide`	division - pushes quotient and remainder
`andop`	logical and, zero is false, anything else is true
`or`	logical or
`not`	logical not
`dup`	copy the number on the top of the operand stack
`exch`	exchange the top two numbers on the operand stack
`nth`	copy the nth number down on the operand stack
`gt`	compare the top two numbers on the operand stack
`lt`	compare again (reverse sense to gt)
`eq`	compare for equality
`damage`	get the current damage in the molecule and push on operand stack
`if`	if the top number on the operand stack is true, skip the next five instructions
`die`	commit suicide
`wait`	wait for a molecule matching a given pattern
`getid`	push my species id (four atoms) on the operand stack
`probe`	test to see if the context specified is writeable
`read`	read one atom from the read context
`write`	write one atom to the write context
`reset`	reset the context to the beginning
`push-context`	make a copy of the designated context on the context stack
`detach`	let a child (or attached) molecule free
`push-label-forward`	search forward in the designated context for a label matching the given label atom

`push-label-backward`	search backward
`pop-context`	given a context number, put the context on the top of the context stack in that context slop
`drop-context`	pop the top of the context stack and discard
`dup-context`	copy the context on the top of the context stack
`copy-context`	deep copy the context on the top of the context stack
`exch-context`	exchange the top two contexts on the top of the context stack
`arm-context`	create a context for the designated molecule arm
`endp`	test to see if the context points to the end of an atom stream
`begin`	test to see if the context points to the beginning of an atom stream
`start-molecule`	start a new molecule
`set-pattern`	set the pattern the molecule will wait for
`other-context`	push the context number of the "other context" slot on the operand stack
`write-context`	similarly for the "write context"
`read-context`	similarly for the "read context"
`exec-context`	similarly for the "execution context"
`skip-forward`	skip forward in a context by a given number of bytes
`skip-backward`	skip backward
`clear`	reset and clear out all the contexts
`nop`	no-op

B The Cloner

```
      literal   ; set pattern to all zeros
            0
          dup
          dup
          dup   ; now we have 4 zeros on the stack
  set-pattern   ; stash it
         wait   ; and wait for a match
          dup   ; got a result, but it might be an error
      literal   ; so check, 4 or greater is error
            4
           gt   ; is the answer bigger than the arm number? -¿ top of stack
           if   ; test
          pop   ; greater than three, so an error, pop the error (skip if tos is true)
        clear   ; go directly to GO
          nop   ; third instruction jumped over
          nop   ; fourth instruction jumped over
          nop   ; fifth instruction jumped over
  arm-context   ; get the context associated with it
 read-context   ;
  pop-context   ; and make it the read context
write-context   ;
start-molecule  ; start a new molecule context to context number on top of stack.
 exec-context   ; where to jump to do the loop
 push-context   ; save it on the context stack
         read   ; read next atom from read context, put on top of stack
        write   ; write next atom to write context
 read-context   ; at the end of the read context?
         endp   ;
           if
                ; test to see if done
 copy-context   ; dup the context on the top of stack
  exec-context  ; copy that context to the execution context
  pop-context   ;
          nop   ;
          nop   ;
write-context   ; end of skip for if.
```

```
      detach   ;
read-context   ;
      detach   ; set newly built context free
       reset   ; and back to the beginning
```

C The Reaper

The code that follows is the assembler for the reaper. It is almost identical at the start to the cloner. It pushes four zeros for the wait pattern, then waits. When it attaches to a molecule it inserts a "die" instruction at the front of the molecule so that when that molecule executes a reset the "die" instruction will be the next to be executed. One question that remains to be investigated is whether other molecules will learn to avoid "reset" in a simulation with reapers.

In this listing the first 17 instructions are skipped as they are exactly the same as the first 17 instructions of the cloner (above).

```
                ;  first 17 instructions skipped
  arm-context
write-context
  pop-context
 exec-context
 push-context
write-context
        probe
           if
 copy-context
 exec-context
  pop-context
          nop
          nop
      literal
          die
        write
write-context
       detach
        reset
```

References

[1] Thomas S. Ray. An approach to the synthesis of life. 1992.