Avida: A Software Platform for Research in Computational Evolutionary Biology

Charles Ofria*
Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824
ofria@msu.edu

Claus O. Wilke
Digital Life Laboratory 136-93
California Institute of Technology
Pasadena, CA 91125
wilke@caltech.edu

Abstract Avida is a software platform for experiments with self-replicating and evolving computer programs. It provides detailed control over experimental settings and protocols, a large array of measurement tools, and sophisticated methods to analyze and post-process experimental data. We explain the general principles on which Avida is built, as well as its main components and their interactions. We also explain how experiments are set up, carried out, and analyzed.

Keywords
Digital organisms, experimental evolution, self-replicating computer programs

1 Introduction

When studying evolution, we have to overcome a large obstacle: Evolution happens extremely slowly. Traditionally, evolution has therefore been a field dominated by observation and theory, though the domestication of plants and animals is often regarded as an early, unwitting evolution experiment [6]. Realistically, we can carry out controlled evolution experiments only with organisms that have very short generation times, so that populations can undergo hundreds of generations within a time frame of months to a few years. With the advances in microbiology, such experiments in evolution have become feasible with bacteria and viruses [18, 10].

However, even with microorganisms, evolution experiments still take a lot of time to complete, and are often cumbersome. In particular, certain data can be difficult or impossible to obtain, and it is often impractical to carry out enough replicas for high statistical accuracy. According to Daniel Dennett, "...evolution will occur whenever and wherever three conditions are met: replication, variation (mutation), and differential fitness (competition)" [7]. It seems to be an obvious idea to set up these conditions in a computer, and to study evolution in silico rather than in vitro. In a computer, it is easy to measure any quantity of interest with arbitrary precision, and the time it takes to propagate organisms for several hundred generations is only limited by the processing power available. In fact, population geneticists have long been carrying out computer simulations of evolving loci, in order to test or augment their mathematical theories (see [15, 13, 11] for some recent examples). However, the assumptions put into these simulations typically mirror exactly the assumptions of the analytical calculations. Therefore, the simulations can be used only to test whether the analytic calculations are error-free, or whether stochastic effects cause a system to deviate from its deterministic description; they cannot test the model assumptions on a more basic level.

An approach to studying evolution that lies somewhere between evolution experiments with biochemical organisms and standard Monte Carlo simulations is the study

* Corresponding author.

© 2004 Charles Ofria and Claus O. Wilke

of self-replicating and evolving computer programs (digital organisms). Digital organisms can be quite complex, and can interact in a multitude of different ways with their environment or each other, so that their study is not a simulation of a particular evolutionary theory but becomes an experimental study in its own right. In recent years, research with digital organisms has grown substantially ([2, 21–23, 12, 9]; see [20] for a recent review), and it is being increasingly accepted by evolutionary biologists [14]. (However, as Barton and Zuidema [3] note, general acceptance will ultimately hinge on whether artificial life researchers embrace or ignore the large body of population genetics literature.) Avida is arguably the most advanced software platform to study digital organisms to date, and is certainly the one that has had the biggest impact in the biological literature so far. Having reached version 2.0, it now supports detailed control over experimental settings; a sophisticated system to design and execute experimental protocols; a multitude of possibilities for organisms to interact with their environment, including depletable resources and conversion from one resource into another; and a module to post-process data from evolution experiments, including tools to find the line of descent from final organisms to their ultimate ancestor, to carry out knockout studies with organisms, and to align and compare organisms' genomes.

1.1 History of Digital Life

The best-known intersection of evolutionary biology with computer science is the genetic algorithm with its many variants (genetic programming, evolutionary strategies, and so on). All these variants boil down to the same basic recipe: (1) create random potential solutions, (2) evaluate each solution, assigning it a fitness value to represent its quality, (3) select a subset of solutions, using fitness as a key criterion, (4) vary these solutions by making random changes or recombining portions of them, (5) repeat from step (2) until a solution is found that is sufficiently good.

This technique turns out to be an excellent method for solving problems, but it ignores many aspects of natural living systems. Most notably, natural organisms must replicate themselves, as there is no external force to do so; therefore their ability to pass their genetic information on to the next generation is the final arbiter of their fitness. Furthermore, organisms in a natural system have the ability to interact with their environment and with each other in ways that are excluded from most algorithmic applications of evolution.

Work on more naturally evolving computational systems began in 1990, when Steen Rasmussen was inspired by the computer game *core war* [8]. In this game, programs are written in a simplified assembly language and made to compete in the simulated core memory of a computer. The winning program is the one that manages to shut down all processes associated with its competitors. Rasmussen observed that the most successful of these programs were the ones that replicated themselves, so that if one copy were destroyed, others would still persist. In the original core war game, the diversity of organisms could not increase, and hence no evolution was possible. Rasmussen then designed a system similar to core war in which the command that copied instructions was flawed and would sometimes write a random instruction instead on the one intended [16]. This flawed copy command introduced *mutations* into the system, and thus the potential for evolution. Rasmussen dubbed his new program the *core world*, created a simple self-replicating ancestor, and let it run.

Unfortunately, this first experiment failed. Though the programs seemed to evolve initially, they soon started to copy code into each other, to the point where no proper self-replicators survived—the system collapsed into a nonliving state. Nevertheless, the dynamics of this system turned out to be intriguing, displaying the partial replication of fragments of code, and repeated occurrences of simple patterns.

The first successful experiment with evolving populations of self-replicating computer programs was performed the following year. Thomas Ray at the University of Delaware designed a program of his own with significant, biologically inspired modifications. The result was the Tierra system [17]. In Tierra, digital organisms must allocate memory before they have permission to write to it, which prevents stray copy commands from killing other organisms. Death occurs only when memory fills up, at which point the oldest programs are removed to make room for new ones to be born.

The first Tierra experiment was initialized with an ancestral program that was 80 lines long. The program filled up the available memory with copies of itself, many of which had mutations that caused a loss of functionality. Yet other mutations were actually neutral and did not affect the organism's ability to replicate—and a few were even beneficial. In this initial experiment, the only selective pressure on the population was for the organisms to increase their rate of replication. Indeed, Ray witnessed that the organisms were slowly shrinking the length of their genomes, since a shorter genome meant that there was less genetic material to copy, and thus it could be copied more rapidly.

This result was interesting enough on its own. However, other forms of adaptation, some quite surprising, occurred as well. For example, some organisms were able to shrink further by removing critical portions of their genome, and then use those same portions from more complete competitors, in a technique that Ray noted was a form of parasitism. Arms races transpired where hosts evolved methods of eluding the parasites, and they, in turn, evolved to get around these new defenses. Some would-be hosts, known as hyper-parasites, even evolved mechanisms for tricking the parasites into aiding them in the copying of their own genomes. Evolution continued in all sorts of interesting manners, making Tierra seem like a choice system for experimental evolution work.

In 1992, Chris Adami began research on evolutionary adaptation with Ray's Tierra system. His intent was to get these digital organisms to evolve solutions to specific mathematical problems, without forcing them use a predefined approach. His core idea was the following: If he wanted a population of organisms to evolve, for example, the ability to add two numbers together, he would monitor organisms' input and output numbers. If an output ever was the sum of two inputs, the successful organisms would receive extra CPU cycles as a bonus. As long as the number of extra cycles was greater than the time it took the organism to perform the computation, the left-over cycles could be applied toward the replication process, providing a competitive advantage to the organism. Sure enough, Adami was able to get the organisms to evolve some simple tasks, but he faced many limitations in trying to use Tierra to study the evolutionary process.

In the summer of 1993, Charles Ofria and C. Titus Brown joined Adami to develop a new digital life software platform, the Avida system. Avida was designed to have detailed and versatile configuration capabilities, along with high precision measurements to record all aspects of a population. Furthermore, whereas organisms are executed sequentially in Tierra, the Avida system simulates a parallel computer, wherein all organisms are executed effectively simultaneously. Since its inception, Avida has had many new features added to it, including a sophisticated environment with localized resources, an events system to schedule actions to occur over the course of an experiment, multiple types of CPUs to form the bodies of the digital organisms, and a sophisticated analysis mode to post-process data from an Avida experiment. Avida is still under active development both at Michigan State University, led by Charles Ofria, and at the California Institute of Technology, led by Claus Wilke.

1.2 Overview of Avida

The Avida software is composed of three main modules. The first is the *Avida core*, which maintains a population of digital organisms (each with its own genome, virtual

hardware, etc.), an environment that maintains the reactions and resources with which the organisms interact, a scheduler to allocate CPU cycles to the organisms, and various data collection objects. The second module is the *graphical user interface* (GUI), which the researcher can use to interact with the rest of the Avida software. The final component is a collection of *analysis and statistics* tools, including a test environment to study organisms outside the population, data manipulation tools to rebuild phylogenies and examine lines of descent, mutation and local fitness analysis tools, and many others, all bound together in a simple scripting language. A fourth module, an interactive help and documentation system, is currently under development.

In the following sections, we will discuss the two modules of Avida that are relevant for experiments with digital organisms, that is, the Avida core and the analysis and statistics tools. The graphical user interface is not treated in this article.

2 Avida Organisms

In Avida, each digital organism is a self-contained computing automaton that has the ability to construct new automata. The organism is responsible for building the genome (computer program) that will control its offspring automaton, and handing that genome to the Avida world. Avida will then construct virtual hardware for the genome to be run on, and determine how the new organism should be placed in the population. The specifics of this process are controlled by the user, as described in detail in the next section. In a typical Avida experiment, a successful organism attempts to make an identical copy of its own genome, and Avida randomly places that copy in the population, killing the previous occupant at that position.

In principle, the only assumption made about these self-replicating automata in the core Avida software is that their initial state can be described by a string of symbols (their genome) and that they autonomously produce offspring organisms. However, in practice our work has focused on automata with a simple von Neumann architecture that operate on an assembly-like language inspired by the Tierra system. Future research projects will likely have us implement additional organism instantiations to allow us to explore additional biological questions.

In the following subsections, we describe the default hardware of our virtual computers, and explain the principles of the language these machines work on.

2.1 Virtual Hardware

The structure of a virtual machine in Avida is depicted in Figure 1. The core of the machine is the central processing unit (CPU), which processes each instruction in the genome and modifies the states of its components appropriately. Mathematical operations, comparisons, and so on can be done on three registers, `AX`, `BX`, and `CX`. These registers each store and manipulate data in in the form of a single 32-bit number. The registers behave identically, but different instructions may act on different registers by default (see Section 2.2). The CPU also has the ability to store data in two stacks. Only one of the two stacks is active at a time, but it is possible to switch the active stack, so both stacks are accessible.

The program memory is initialized with the genome of the organism. Execution begins with the first instruction in memory and proceeds sequentially: Instructions are executed one after the other, unless an instruction (such as a jump) explicitly interrupts sequential execution. Technically, the memory space is organized in a circular fashion, such that after the CPU executes the last instruction in memory, it will loop back and continue execution with the first instruction again. However, at the same time the memory has a well-defined starting point, important for the creation of offspring organisms.

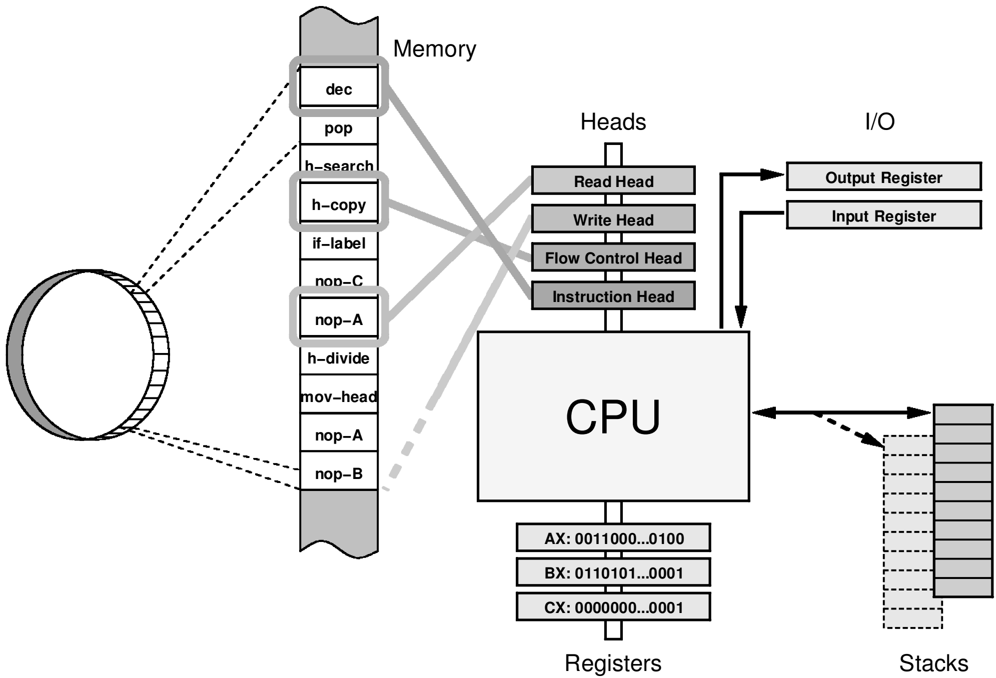

Figure 1. The standard virtual hardware in Avida: CPU, registers, stacks, heads, memory, and I/O functionality.

Somewhat out of the ordinary in comparison with standard von Neumann architectures are the four CPU components called *heads*. Heads are essentially pointers to locations in the memory. They remove the need of absolute addressing of memory positions, which makes the evolution of programs more robust to size changes that would otherwise alter these absolute positions. Among the four heads, only the instruction pointer exists in standard computer architectures. It identifies the instruction currently being executed by the CPU. The instruction pointer moves one instruction forward when the execution of the previous instruction has been completed, unless that instruction has specifically moved the instruction pointer elsewhere.

The other three heads—read head, write head, and flow control head—are unique to the Avida virtual hardware. The read and write heads are used in the self-replication process. In order to generate a copy of its genome, an organism must have a means of reading instructions from memory and writing them back to a different location. The read head indicates the position in memory from which instructions are currently being read, and the write head likewise indicates the position to which instructions are currently being written. The positions of all four heads can be manipulated with special commands. In that way a program can position the read and write heads appropriately in order to self-replicate.

The flow control head is used for jumps and loops. Several commands will reposition the flow control head, and other commands will move specific heads to the same position in memory as the flow control head.

Finally, the virtual machines have an input buffer and an output buffer, which they use to interact with their environment. The way in which this communication works is that the machines can read in one or several numbers from the input buffer, perform computations on these numbers with the help of the internal registers `AX`, `BX`, `CX`, and the stacks, and then write the results to the output buffer. This interaction with

the environment plays a crucial role in the evolution of Avida organisms, and will be explained in detail in Section 3.3.

2.2 Genetic Language

In the previous subsection, we described the elements that constitute the virtual hardware of an Avida organism. Now we turn to the elements of the assembly-like language that make up the genomes and in turn the memory of the hardware.

It is important to understand that there is not a single Avida language. Instead, we have a collection of different languages. The virtual hardware in its current form can execute hundreds of different instructions, but only a small fraction of them are used in a typical experiment. The instructions are organized into subsets of the full range of instructions. We call these subsets *instruction sets*. Each instruction set forms a logical unit and can be considered a programming language.

Each instruction in Avida has a well-defined function in any context, that is, there are no syntactically incorrect programs. Instructions do not have arguments per se, but the behavior of certain instructions can be modified by succeeding instructions in memory. A genome is therefore nothing more than a sequence of symbols in an alphabet composed of the instruction set, as DNA is a sequence made up of the 4 nucleotides, or proteins are sequences made up of the 20 amino acids.

Here, we will give an overview of the default instruction set, which contains 26 instructions. This set is explained in more detail in Appendix 1, as well as an alternative instruction set that has been used in multiple research projects and does not make use of the heads.

2.2.1 Template Matching and Heads

One important ingredient of most Avida languages is the concept of template matching. Template matching is a method of indirectly addressing a position in memory. This method is similar to the use of labels in many programming languages: Labels tag a position in the program, so that jumps and function calls always go to the correct place, even when other portions of the source code are edited. The same reasoning applies to Avida genomes, because mutations may cause insertions or deletions of instructions that shift the position of code and would otherwise jeopardize the positions referred to. Since there are no arguments to instructions, positions in memory are determined by series of subsequent instructions. We refer to a series of instructions that indicates a position in the genome as a *template*.

Template based addressing works as follows. When an instruction is executed that must reference another position in memory, subsequent *nop* instructions (described below) are read in as the template. The CPU then searches linearly through the genome for the first occurrence of the complement to this template, and uses the end of the complement as the position needed by the instruction. Both the direction of the search (forward or backward from the current instruction) and the behavior of the instruction if no complement is found are defined specifically for each instruction.

Avida templates are constructed out of no-operation (nop) instructions, that is, instructions that do not alter the state of either CPU or memory when they are directly executed. There are three template-forming nops: `nop-A`, `nop-B`, and `nop-C`. They are cyclically complementary, that is, the complement of `nop-A` is `nop-B`, the complement of `nop-B` is `nop-C`, and the complement of `nop-C` is `nop-A`. A template is composed of consecutive nops only. A template will end with the first non-nop instruction.

We give an example in Figure 2. Apart from the nop instructions, it contains the command `jump-f`, which causes the CPU to inspect the template immediately following the `jump-f` command and to search for the complement template in the forward direction (likewise, the command `jump-b` would initiate a search in the backward direction).

	⋮	Some code.
10	`jump-f`	Here comes the jump. Search for the complement of the template in forward direction.
11	`nop-A`	This is the template. Let's call it α.
12	`nop-B`	
13	`pop`	Some other code that is skipped.
	⋮	
17	`nop-B`	The complement template $\bar{\alpha}$.
18	`nop-C`	
	⋮	The program continues ...

Figure 2. Example code demonstrating nop templates and flow control.

	⋮	Some code.
10	`h-search`	Prepare the jump by placing the flow control head at the end of the complement template in forward direction.
11	`nop-A`	This is the template. Let's call it α.
12	`nop-B`	
13	`mov-head`	The actual jump. Move the flow control head to the position of the instruction head.
14	`pop`	Some other code that is skipped.
	⋮	
18	`nop-B`	The complement template $\bar{\alpha}$.
19	`nop-C`	
	⋮	The program continues ...

Figure 3. Example code demonstrating flow control with heads based instruction set.

Now it becomes a bit more complicated. The example in Figure 2 is actually from an old instruction set that was used for a long time but is not in use by default anymore (see Section 1 in Appendix 1). The new default instruction set does not contain the instruction `jump-f` or `jump-b`. Instead, jumps have to be implemented through clever manipulation of the different heads. This happens in two stages. First, the instruction `h-search` is used to position the flow control head at the desired position in memory. Then, the instruction head is moved to that position with the command `mov-head`. Figure 3 shows how the previous example can be coded with these commands.

Although this example looks somewhat awkward at first glance, evolution of control structures such as loops is facilitated in the heads based instruction set. In order to loop over some piece of code, it is only necessary to position the flow control head correctly once, and to have the command `mov-head` at the end of the block of code that should be looped over. Since there are several ways in which the flow control head can be positioned correctly, of which the above example is only a single one, there are many more ways in which loops can be generated than in the corresponding case with `jump-f` and `jump-b`. One construction that is not possible with normal jump instructions, for example, is a loop with varying entry points. The position of the flow

01	pop	We assume the stack is empty. In that case, the pop returns 0, which is stored in BX.
02	pop	Write 0 into the register AX as well.
03	nop-A	
04	inc	Increment BX.
05	inc	Increment AX.
06	nop-A	
07	inc	Increment AX a second time.
08	nop-A	
09	swap	The swap command exchanges the contents of a register with the one of its complement register. Followed by a nop-C, it exchanges the contents of AX and CX. Now, BX= 1, CX= 2, and AX is undefined.
10	nop-C	
11	add	Now add BX and CX and store the result in AX.
12	nop-A	The program continues with BX= 1, CX= 2, and AX= 3.
	⋮	

Figure 4. Example code demonstrating the principle of nop modification.

control head could be modified in the block of code that is looped over. In that way, after every execution of that block, a different section of code would be executed the next time. As a consequence, the heads based method can lead to more complicated flow control than the relatively rigid direct jumping to templates.

2.2.2 Nops as Modifiers

The instructions in the Avida programming language do not have arguments in the usual sense. However, as we have seen above for the case of template matching, the effect of certain instructions can be modified if they are immediately followed by nop instructions. A similar concept exists for operations that access registers. The `inc` instruction, for example, increments a register by one. If `inc` is not followed by any nop, then by default it acts on the BX register. However, if a nop is present immediately after the `inc`, then the register on which `inc` acts is specified by the type of the nop. For example, `inc nop-A` increments the AX register, and `inc nop-C` the CX register. Of course, `inc nop-B` increments the BX register, and hence works identically to a bare `inc` command. Similar nop modifications exist for a range of instructions, including those that perform arithmetic such as `inc` or `dec`, stack operations such as `push` or `pop`, and comparisons such as `if-n-equ`. The details can be found in Appendix 1. For some instructions that work on two registers, in particular comparisons, the concept of the complement nop is important, because the two registers are specified in this way. Similarly to the nops in the template matching, registers are cyclically complementary to each other, that is, BX is the complement to AX, CX to BX, and AX to CX. The instruction `if-n-equ`, for example, acts on a register and its complement register. By default, `if-n-equ` determines whether the contents of the BX and CX registers are identical. However, if `if-n-equ` is followed by a `nop-A`, then it will compare AX and BX. Figure 4 shows a piece of example code that demonstrates the principles of nop modification and complement registers.

Nop modification is also necessary for the manipulation of heads. The instruction `mov-head`, for example, by default moves the instruction head to the position of the flow control head. However, if it is followed by either a `nop-B` or a `nop-C`, it moves the read head or the write head, respectively. A `nop-A` following a `mov-head` leaves the default behavior unaltered.

2.2.3 Memory Allocation and Division

When a new Avida organism is created, the CPU's memory is exactly the size of its genome, that is, there is no additional space that the organism could use to store its offspring-to-be as it makes a copy of its program. Therefore, the first thing an organism has to do at the start of self-replication is to allocate new memory. In the default instruction set, memory allocation is done with the command `h-alloc`. This command extends the memory by the maximal size that a child organism is allowed to have. As we will discuss later, there are some restrictions on how large or small a child organism is allowed to be in comparison with the parent organism, and the restriction on the maximum size of a child organism determines the amount of memory that `h-alloc` adds. The allocation happens always at a well-defined position in the memory. As we mentioned in Section 2.1, although the memory is considered to be cyclical in the sense that the CPU will continue with the first instruction of the program once it has executed the last one, the virtual machine nevertheless keeps track of which instruction is the beginning of the program, and which is the end. By default, `h-alloc` (as well as all alternative memory allocation instructions, such as the old `allocate`) insert the new memory between the end and the beginning of the program. After the insertion, the new end is at the end of the inserted memory. The newly inserted memory is initialized either to a default instruction, typically `nop-A`, or to random code, depending on the choice of the experimenter.

Once an organism has allocated memory, it can start to copy its program code into the newly available memory block. This copying is done with the help of the control structures we have already described, in conjunction with the instruction `h-copy`. This instruction copies the instruction at the position of the read head to the position of the write head and advances both heads. Therefore, for successful self-replication an organism mainly has to assure that initially, the read head is at the beginning of the memory and the write head is at the beginning of the newly allocated memory, and then it has to call `h-copy` the correct number of times.

After the self-replication has been completed, an organism issues the `h-divide` command, which splits off the instructions between the read head and the write head, and uses them as the genome of a new organism. The new organism is handed to the Avida world, which takes care of placing it in a suitable environment, and so on. If there are instructions left between the write head and the end of the memory, these instructions are discarded, so that only the part of the memory from the beginning to the position of the read head remains after the divide.

In most natural asexual organisms, the process of division results in organisms literally splitting in half, effectively creating two offspring. Thus, the default behavior of Avida is to reset the state of the parent's CPU after the divide, turning it back into the state it was in when it was first born. In other words, all registers and stacks are cleared, and all heads are positioned at the beginning of the memory. The full allocation and division cycle is illustrated in Figure 5.

Not all `h-divide` commands that an organism issues lead necessarily to the creation of an offspring organism. There are a number of conditions that have to be satisfied; otherwise the command will fail. (Failure of a command means essentially that the command is ignored, while a counter keeping track of the number of failed commands in an organism is incremented. It is possible to configure Avida to punish organisms

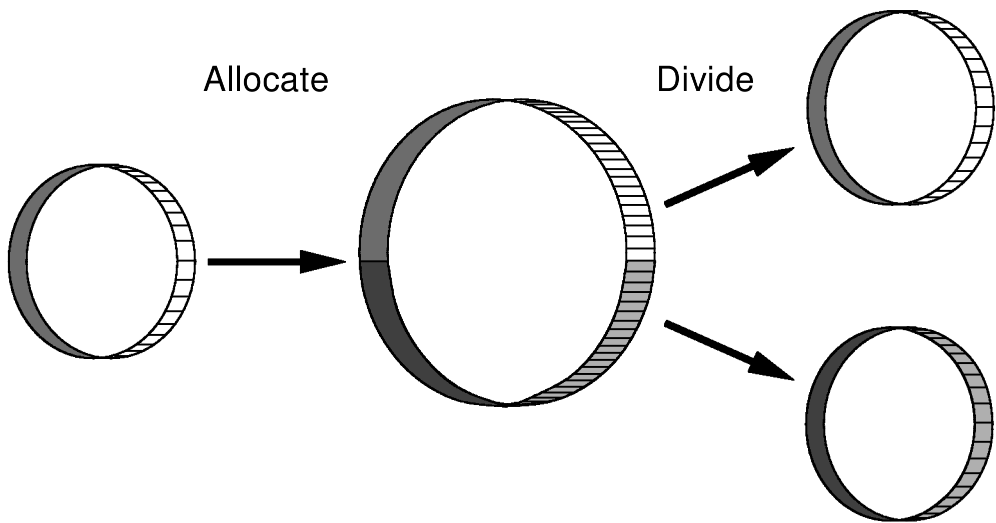

Figure 5. The `h-alloc` command extends the memory, so that the program of the child organism can be stored. Later, on `h-divide`, the program is split into two parts, one of which turns into the child organism.

with failed commands.) The following conditions are in place: An `h-divide` fails if either the parent or the offspring would have less than 10 instructions, the parent has not allocated memory, less than half of the parent was executed, less than half of the offspring's memory was copied into, or the offspring would be too small or too large (as defined by the experimenter).

2.3 Mutations

So far, we have described all the elements that are necessary for self-replication. However, self-replication alone is not sufficient for evolution. There must be a source of variation in the population, which comes from random *mutations*.

The main form of mutations in Avida are so-called copy mutations, which arise through erroneously copied instructions. Such miscopies are a built-in property of the instruction `h-copy`. With a certain probability, chosen by the experimenter, the command `h-copy` does not properly copy the instruction at the location of the read head to the location of the write head, but instead writes a random instruction to the position of the write head. It is important to note that the instruction written will always be a legal one, in the sense that the CPU can execute it. However, the instruction may not be meaningful in the context in which it is placed in the genome, which in the worst case can render the offspring organism nonfunctional.

Another commonly used kind of mutation comprises insertion and deletion mutations. These mutations are applied on `h-divide`. After an organism has successfully divided off a child organism, an instruction in the child's memory may by chance be deleted, or a random instruction may be inserted. The probabilities with which these events occur are again determined by the experimenter. Insertion and deletion mutations are useful in experiments in which frequent changes in genome size are desired.

Next, there are point (or cosmic ray) mutations. These mutations affect not only organisms as they are being created (like the other types described above), but all living organisms. Point mutations are random changes in the memory of the virtual machines. One of the consequences of point mutations is that a program may change while it is being executed. In particular, the longer a program runs, the more sus-

ceptible it becomes to point mutations. This is in contrast to copy or insertion and deletion mutations, whose impact depends only on the length of the program, not on the execution time.

Finally, it is important to note that organisms in Avida can also have *implicit* mutations. Implicit mutations are modifications in a child's program that are not directly caused by any of the external mutation mechanisms described above, but rather by an incorrect copy algorithm of the parent organism. For example, the copy algorithm might skip some instructions of the parent program, or copy a section of the program twice (effectively a gene duplication event). Another example is an incorrectly placed read head or write head on divide. Implicit mutations are the only ones that cannot easily be controlled by the experimenter. They can, however, be turned off completely by using the FAIL_IMPLICIT option in the genesis file. With this option, any offspring organism whose genome has differences to its parent genome that were not caused by explicit mutations is discarded, rather than placed into the population.

2.4 Phenotype

Each organism in an Avida population has a phenotype associated with it. Phenotypes of Avida organisms are defined in the same way as they are defined for organisms in the natural world: The phenotype of an organism comprises all observable characteristics of that organism. As an organism in Avida goes through its life cycle, it will self-replicate and, at the same time, interact with the environment. The primary mode of environmental interaction is by inputting numbers from the environment, performing computations on those numbers, and outputting the results. The organisms receive a benefit for performing specific computations associated with resources (see Section 3.3 below).

In addition to tracking computations, the phenotype also monitors several other aspects of the organism's behavior, such as its gestation length (the number of instructions it executes to produce an offspring, often also called *gestation time*), its age, whether it has been affected by any mutations, how it interacts with other organisms, and its overall fitness. These data are used both to determine how many CPU cycles should be allocated to the organism and for various statistical purposes.

2.5 Genotype and Species

In Avida, organisms are classified into several taxonomic levels. The lowest taxonomic level is called *genotype*. All organisms that have exactly the same initial genomes are considered to have the same genotype. Certain statistical data are collected only at the genotype level. We pay special attention to the most abundant genotype in the population—the *dominant* genotype—as a method of determining what the most successful organisms in the population are capable of. If a new genotype is truly more fit than than the dominant one, organisms with this higher fitness will rapidly take over the population.

We classify a genotype as *threshold* if there are three or more organisms that have ever existed with that genotype (again, the value three is not hard-coded, but configurable by the experimenter). Often, deleterious mutants appear in the population. These mutants are effectively dead and disappear again in short order. Since these mutants are not able to self-replicate successfully (or at least not well), there is a low probability of them reaching an abundance of three. Thus for any statistics we want to collect about the *living* portion of the population, we focus on those organisms whose genotype has the threshold characteristic.

We also consider a layer of taxonomic classification one step above the genotype, which we equate to *species*. In the natural world, biologists classify sexual organisms as part of the same species if they mate in the wild and produce fertile offspring. The

organisms in Avida are primarily asexual; therefore the standard biological definition does not translate perfectly, but unfortunately there is no consensus on the definition of asexual species. As such, we aim to indirectly incorporate the sexual concept of species into Avida.

Each species in Avida has a prototype associated with it that is the initial genotype that formed the new species. This genotype is used to determine what other genotypes are going to be part of the species. When a genotype reaches threshold, we compare it against existing species with a two-phase test. If the genotype passes both parts, we label it as part of that species. If it fails either part for all existing species, it is classified as a new species. In the first part of this test, we make sure that the genotype being tested matches in key aspects with the phenotype associated with the species prototype; all genotypes in a species must be phenotypically identical in their environmental interactions (portions of the phenotype such as age are ignored). In the second part of the test, we create hybrid genotypes by crossing over the genotype being tested with the prototype of the species at all possible crossover points. We then take each of these hybrids and run them in a test environment (that does not feed back into the main Avida world in any way) to determine if the hybrids are viable. If most of them are viable (the exact threshold of viability being set by the experimenter), then the genotype passes this phase of the test.

Note that by default species classification is turned off in Avida, because it is CPU intensive to perform all of these tests. A limited version of it is also available where each genotype is only tested against the species of its parent, not all species in the population.

Species and genotype handling are all controlled by the genebank object in the Avida source code.

3 Avida World

In the previous section, we have presented the details of the inner workings of Avida organisms. In this section, we will expand that view by explaining how the environment in which these organisms thrive is structured, and how they interact with it.

In general, the Avida world has a fixed number N of positions, or *cells*. Each cell can be occupied by exactly one organism, so that the maximum population size at any given time is N. Each of these organisms is being run on a virtual CPU, and some of them may be running faster than others. Avida has a scheduler (see Section 3.1) that divides up time from the real CPU so that these virtual CPUs execute in a simulated parallel fashion.

While an Avida organism runs, it may interact with the environment (Section 3.3) or other organisms (Section 3.4). When it finally reproduces, it hands its offspring organism to the Avida world, which places the newborn organism in either an empty or an occupied cell, according to rules we describe in Section 3.2. If the offspring organism is placed in an already occupied cell, the organism currently occupying that cell is killed and removed, irrespective of whether it has already reproduced or not.

3.1 Scheduling

In the simplest of Avida experiments, all virtual CPUs run at the same speed. This method of time sharing is simulated by executing one instruction on each of the N virtual CPUs in order, then starting over to execute a second instruction on each one, and so on. An *update* in Avida is defined as a point where the average organism has executed k instructions (where $k = 30$ by default). In this simple case, for one update we carry out k rounds of execution.

In more complex environments, however, the situation is not so trivial. Different organisms will have their virtual CPUs running at different speeds (the details of which

are described in Section 3.3), and the scheduler must portion out cycles appropriately to simulate that all CPUs are running in parallel. Each organism has associated with it a value called *merit*. The merit indicates how fast the CPU should run. Merit is a unitless quantity, and is only meaningful when compared with the merits of other organisms. Thus, if organism A has twice the merit of organism B, then A should execute twice as many instructions in any given time frame as B.

Avida handles this with two different schedulers. The first one is a perfectly integrated scheduler, which comes as close as possible to portioning out CPU cycles in proportion to merit. Obviously, only whole time steps can be used; therefore perfect proportionality is not possible in general for small time frames. For time frames long enough such that the granularity of individual time steps can be neglected, the difference between the number of cycles given to an organism and the number of cycles the organism should receive at its merit is negligible.

The second scheduler is probabilistic. At each point in time, the next organism to be selected is chosen at random, but with the probability of an individual being chosen proportional to its merit. Thus *on average* this scheduler is perfect, but there are no guarantees.

In practice the perfectly integrated scheduler is faster, but occasionally can cause odd effects, because it is possible for the organisms to become synchronized, particularly at low mutation rates where a single genotype can represent a large portion of the population. The stochastic scheduler may be preferred for projects where this effect might be a problem. By default, Avida uses the perfectly integrated scheduler.

3.2 World Topologies and Birth Methods

The N cells of the Avida world can be assembled into different topologies that affect how offspring organisms are placed and how organisms interact (as described in Section 3.4). Currently, there are two world topologies: a 2D grid with Moore neighborhood (each cell has eight neighbors) and a fully connected (sometimes called *well-stirred* or *mass action*) topology. In the fully connected topology each cell is a neighbor to every other cell. New topologies can easily be implemented by listing the neighbors associated with each cell (though more work might need to be done in the user interface to properly visualize the topology).

When a new organism is about to be born, it will replace either the parent cell or another cell from the neighborhood. The specifics of this placement are set up by the experimenter. The two most commonly used methods are *replace random*, which chooses randomly from the neighborhood, and *replace oldest*, which picks the oldest organism from the neighborhood to replace (with a preference for empty cells if any exist).

Fully connected topologies are used in analogy with experiments with microbes in well-stirred flasks or chemostats. These setups allow for exponential growth of new genotypes with a competitive advantage, so that transitions in the state of the population can happen rapidly. Local neighborhoods, on the other hand, are more akin to a Petri dish, and the spatial separation between different organisms puts limits on growth rates and allows for a slightly more diverse population [4].

In choosing which organism in a neighborhood to replace, random placement matches up well with the behavior of a chemostat, where a random portion of the population is continuously drawn out to keep population size constant. Experiments have shown [1], however, that evolution occurs more rapidly when the oldest organism in a neighborhood is the first to be killed off. In such cases, all organisms are given approximately the same chance to prove their worth, whereas in random replacement, about half the organisms are killed before they have the opportunity to produce a single offspring. Interestingly, when replace oldest is used in 2D neighborhoods, 40% of the

time it is the parent that is killed off. This observation makes sense, because the parent is necessarily old enough to have produced at least one offspring.

Note that in the default setup of Avida, the only way for an organism to die is for it to be replaced by another organism being born. It is also possible to enable an independent death method that will kill off an organism after it has executed a specified number of instructions, which can either be a constant or be proportional to the organism's genome length. In some cases, if death is not enabled, a population can lose all ability to self-replicate, but persist because organisms have no way of being purged. This effect can lead to confusing experimental results if the cause is not identified.

3.3 Environment and Resources

All organisms in Avida are provided with the ability to absorb a default resource that gives them their base merit. An Avida environment can, however, contain other resources that the organisms can absorb to modify their merit. The organisms absorb a resource by carrying out the corresponding computation, or *task*.

An Avida environment is described by a set of resources and a set of reactions that can be triggered to interact with those resources. A reaction is defined by a computation that the organism must perform to trigger it, a resource that is consumed by it, a merit effect on the organism (which can be proportional to the amount of resource absorbed or available), and a by-product resource if one should be produced. Reactions can also have restrictions associated with them that limit when a trigger will be successful. For example, another reaction can be required to have been triggered first, or a limit can be placed on the number of times an organism can trigger a certain reaction.

A resource is described by an initial quantity (which can be infinite if a resource should not be depletable), an inflow rate (the amount of that resource that should come into the population per update), and an outflow rate (the fraction of the resource that should be removed each update.) If resources are made to be depletable, then the more organisms trigger a reaction, the less of that resource is available for each of them. This setup allows multiple, diverse subpopulations to coexist stably in an Avida world [5].

The default Avida environment rewards nine boolean logic operations, each associated with a non-depletable resource, but organisms can receive only one reward per computation. Other pre-built environments that come with Avida include one with 78 different logic operations rewarded; one similar to the default nine-resource environment, but with the resources set up to be depletable, with fixed inflow and outflow rates; and one with nine computations rewarded, and where only the resources associated with the simplest computations have an inflow into the system, and those for more complex operations are produced as by-products, in sequence, from the reactions using up resources associated with simpler computations.

An important aspect of Avida is that the environment does not care *how* a computation is performed, only that the output of the organism being tested is correct given the inputs it took in. As a consequence, the organisms find a wide variety of ways of computing their outputs, some of which can be surprising to a human observer, seeming to be almost inspired.

Even though organisms can carry out tasks and collect rewards at any time in their gestation cycle, these rewards do not immediately affect the speed at which their CPU runs. The CPU speed (merit) is set only once, at the beginning of the gestation cycle, and then held constant until the organism divides. At that point, both the organism and its offspring get a new merit, which reflects the bonuses the organism collected during the gestation cycle it just completed. In a sense, the organisms collect rewards for their offspring, rather than for themselves. The reason why we do not change an organism's merit during its gestation cycle is to level the playing field between old and

young organisms. If organisms were always born with a low initial CPU speed, then they might never execute enough instructions to carry out tasks in the first place. At the same time, mutants specialized in carrying out tasks but not dividing could concentrate all CPU time on them, thus effectively shutting down replication in the population. It can be shown that the average fitness of a population in equilibrium is independent of whether organisms get the bonuses directly or collect them for their offspring [19].

3.4 Organism Interactions

As explained in Section 3.2, populations in Avida have a firm cap on their size, which makes space the fundamental resource that the organisms must compete for. In the simplest Avida experiments, the only interaction between organisms is that an organism is killed when another gives birth, in order to make room for the offspring. In slightly more complex experiments, the organisms are rewarded with a higher merit and hence a larger share of the CPU cycles for performing tasks. Since there are only a fixed number of CPU cycles given out each update, the competition for them becomes a second level of indirect interaction among the organisms. As the environment becomes more complex still, multiple resources take the place of fixed merit bonuses for performing tasks, and the organisms must now compete over each of these resources independently. In the end, however, all these interactions boil down to the indirect competition for space: More resources imply a higher merit, which in turn grants the organisms a larger share of the CPU cycles, allowing them to replicate more rapidly and claim more space for their genotype.

In most Avida experiments, indirect competition for space is the only level of interaction we allow; organisms are not allowed to directly write to or read from each other's genomes, so that Tierra-style parasites cannot form (although the configuration files do allow the experimenter to enable them). The more typical way of allowing parasites in Avida is to enable the `inject` command in the Avida instruction set. This command works similarly to the divide, except that instead of replacing an organism in a target cell, the would-be offspring is inserted into the memory of the organism occupying the target cell; the specific position in memory to which it is placed is determined by the template that follows the `inject`.

In Tierra, parasites can replicate more rapidly than non-parasites, but an individual parasite poses no direct harm to the host whose code it uses. These organisms could, therefore, be thought of more directly as cheaters in the classic biological sense, as they effectively take advantage of the population as a whole. In Avida, a parasite exists directly inside of its host, and makes use of the CPU cycles that would otherwise belong to the host, thereby slowing down the host's replication rate. Depending on the type of parasite, it can either take all of the host's CPU cycles (thereby killing the host) and use them for replicating and spreading the infection, or else spread more slowly by using only a portion of the host's CPU cycles (sickening it), but reducing the probability of driving the hosts, and hence itself, into extinction.

In the future, we plan to implement two other forms of interactions. First, we plan to implement *sensors* with which organisms can detect the presence of resources, which would allow them to exchange chemical signals. Second, we are considering more direct *communication*, whereby the organisms can send numbers to each other, and possibly distribute computations among themselves to solve environmental challenges more rapidly.

4 Experimental Environment

The configuration of an Avida experiment requires setting up five different initialization files. The most important of these is the *genesis* file, which contains a list of variables

that control all of the basic settings of a run, including the population size, the mutation rates, and the names of all of the other configuration files to use. Next, we have the *instruction set*, which describes the specific genetic language used in the experiment. Third is the *ancestral organism* that the population should be seeded with. Fourth, we have the *environment* file that describes which resources are available to the organisms, and defines reactions by the tasks that trigger them, their value, the resource that they use, and any by-products that they produce. The final configuration file is that of *events*, and is used to describe specific actions that should occur at designated time points during the experiment, including most data collection, as well as direct disruptions to the population. Each of these files is described in more detail in the following subsections.

4.1 The Genesis File

The file `genesis` is the main configuration file for Avida. With this file, the experimenter can set up all of the basic conditions for a run. Below are detailed descriptions for some of the most important settings in `genesis`; for the other, less-used settings, refer to the documentation that is shipped with the Avida software.

`WORLD-X WORLD-Y` These settings determine the size of the Avida grid that the organisms populate. In mass action mode the shape of the grid is not relevant, only the number of organisms that are in it.

`RANDOM_SEED` The random number seed initializes the random number generator. This seed is the only value that should be altered in a collection of replicate runs. Setting the random number seed to zero (or a negative number) will base the seed on the starting time of the run—effectively a random number seed.

`INST_SET` The name of the instruction set configuration file to use for the experiment.

`START_CREATURE` The name of the ancestral genotype that should be used to seed the experiment.

`ENVIRONMENT_FILE` The name of the configuration file that describes the environment to be used in the experiment.

`EVENT_FILE` The name of the events file to use for the experiment.

`ANALYZE_FILE` If Avida is started in analyze mode (i.e., Avida carries out post-experimental data analysis rather than an actual experiment), this setting specifies the filename of the script that should be run.

`BIRTH_METHOD` The birth method defines how the placement of a child organism is determined. Currently, there are six birth methods—the first four (0–3) are all grid-based (offspring are only placed in the immediate neighborhood), and the last two (4, 5) assume a well-stirred population. In all nonrandom methods, empty sites are preferred over replacing a living organism.

`CHILD_SIZE_RANGE` This setting defines the maximal difference in genome size between a parent and offspring, and can be used to prevent out-of-control size changes. The default of 2.0 means that the genome of the child must be between one-half and twice the length of the parent. A value of 1.0 will ensure fixed-length organisms. (For fixed-length organisms, insertion and deletion mutations should be turned off as well.)

`COPY_MUT_PROB` Whenever an organism tries to copy a single instruction, the copy process can fail, with the probability given by this parameter. When it fails, that is, when a mutation occurs, a random instruction is copied to the destination. In practice, copy mutations are the most common type of mutations that we use in Avida experiments.

`INS_MUT_PROB, DEL_MUT_PROB` Probabilities with which instructions are inserted or deleted, respectively, per site (i.e., position in the genome) and per generation. Instructions are inserted or deleted only once per gestation cycle, when a new organism is born. Each of these mutations changes the genome length. Deletions just remove an instruction while insertions add a new, random instruction at a random position. Multiple insertions and deletions are possible each generation.

`DIVIDE_INS_PROB, DIVIDE_DEL_PROB` Like `INS_MUT_PROB` and `DEL_MUT_PROB`, except that the probability is now for the insertion or deletion of one instruction per genome (rather than per site) per gestation cycle. With these parameters, multiple insertions and deletions are not possible within a single generation.

`POINT_MUT_PROB` Point mutations (sometimes referred to as *cosmic ray* mutations) occur every update; the value set here is the probability with which a single instruction will be mutated in one update. In other words, the value should be kept low. (The default is zero.) If a mutation occurs, the instruction is replaced with a random instruction. In practice, point mutations slow Avida down, because many random numbers have to be calculated every update.

`TRACK_MAIN_LINEAGE` In a normal Avida run, the genebank keeps track of all existing genotypes, and deletes them when the last organism of that genotype dies out. With this flag set, a genotype will not be deleted unless both it and all of its descendents have died off. This setting allows us to track back from any genotype to its distant ancestors, monitoring all of the differences along the way. The ancestral information can be written out with the event `dump_historic_pop`, explained in Section A2.1 in Appendix 2.

4.2 The Instruction Set File

An instruction set file consists of a list of instructions that belong to that instruction set, each of which is followed by a series of numbers that define how that instruction should be used. These numbers are (in order):

`redundancy` Frequency of the instruction in the set. One instruction with twice the redundancy of another will have twice the probability of being mutated to. A redundancy of zero is allowed, and indicates that organisms injected by the user are allowed to have this instruction, but it can never be acquired through mutations.

`cost` Number of CPU cycles required to execute the instruction. One is the default if this value is not specified.

`ft_cost` Additional cost to be paid the first time the instruction is executed. The cost can be used to lower the diversity of instructions within organisms. The default value here is zero.

`prob_fail` Probability that the instruction will not work properly. If an instruction fails, it will simply do nothing, but still cost the CPU cycles to execute. The default probability of failure is zero.

Refer to Appendix 1 for more information on the specific instructions in the default Avida instruction set.

4.3 The Ancestral Organism

Each Avida run must be seeded with a self-replicating organism, which will fill up the population and be the basis for all further evolution. Avida comes with a collection of available ancestors (such as the one described in Section A1.3 of Appendix 1). Any genotype file output from Avida can be used as an ancestor in a subsequent experiment.

4.4 The Environment File

The environment file controls the resources and reactions available in an Avida run. By default, all resources are globally available to all organisms. (Spatially localized resources are more complicated and will not be described in this document. We are currently working on methods to simplify their use.) A resource is defined by three main characteristics:

`inflow` Number of units of the resource that enter the population over the course of an update, spread evenly throughout that update.

`outflow` Fraction of the resource that will flow out of the population each update. As with inflow, outflow happens continuously over the course of the update.

`initial` The initial abundance of the resource in the population at the start of an experiment.

The following is an example of a `RESOURCE` statement that begins a run with 10,000 units of a resource called "glucose" and sets up a chemostat-like resource flow with an equilibrium concentration of 10,000 units for unused resources:

```
RESOURCE glucose:initial=10000:inflow=100:outflow=0.01
```

At equilibrium, 100 units are flowing out each update (because the resource has an abundance of 10,000 if not used), and 100 units are flowing in.

Reactions are somewhat more complicated than resources. A `REACTION` statement has the following form:

`REACTION` *name task* `process:`*arguments* `requisite:`*arguments*

Here, *name* is an identifier for the reaction, and can be chosen arbitrarily. The *task* identifies which task will trigger the reaction. The following tasks are predefined in Avida: `echo`, `not` (one-input tasks); `add`, `sub` (two-input computational tasks); `nand`, `and`, `orn`, `or`, `andn`, `nor`, `xor`, `equ` (all possible two-input logic tasks); `logic_3AA`, `logic_3AB`, `logic_3AC`, ..., `logic_3CP` (all 68 possible three-input logic tasks). The `process` statement determines consumption of resources, their by-products, and the resulting bonuses, and the `requisite` statement limits under what conditions a reaction can be triggered. Each of the latter two statements has one or several arguments, separated by colons.

We now list the arguments for the `process` statement:

`resource` The name of the resource consumed. By default, an infinite resource is being consumed, which means that the `max` limit (below) determines the amount actually absorbed. Default: `resource=infinite`

`value` The value is multiplied by the amount of the resource consumed to obtain the bonus. (0.5 may be inefficient, while 5.0 is very efficient.) This parameter allows different reactions to make use of the same resource at different efficiency levels. Default: `value=1.0`

`type` Determines how to apply the bonus (i.e., the amount of the resource absorbed times the value of this process) to change the merit of the organism. There are three possible settings: (1) `type=add`—The bonus is added to the current merit. (2) `type=mult`—The current merit is multiplied by the bonus. (Warning: If the bonus is ever less than one, this setting will be detrimental.) (3) `type=pow`—The current merit is multiplied by 2^{bonus}. The last setting is similar to multiplicative, but positive bonuses are always beneficial, and negative bonuses are harmful. Default: `type=add`

`max` The maximum amount of the resource consumed per occurrence. Default: `max=1.0`

`min` The minimum amount of resource required. If less than this quantity is available, the reaction ceases to proceed. Default: `min=0.0`

`frac` The maximum fraction of the available resource that can be consumed. Default: `frac=1.0`

`product` The name of the by-product resource. At the moment, only a single by-product can be produced at a time. Default: `product=none`

`conversion` The rate of conversion to the by-product resource. Default: `conversion=1.0`

If no process is given, a single associated process with all default settings is assumed. If multiple process statements are given, all are executed when the reaction is triggered. If we wanted to set all parameters of the process statement to their default values, we would write:

```
process:resource=infinite:value=1:type=add:max=1:min=0:\
  frac=1:product=none:conversion=1
```

This statement has many redundancies; for example, it indicates that the associated reaction should use the infinite resource, making the `frac` and `min` settings irrelevant. Likewise, since `product` is set to `none`, the `conversion` statement is superfluous.

Finally, the following arguments can be given (in any combination) to the `requisite` statement:

`reaction` Prevents the reaction from being triggered until the other reaction specified here has been triggered first. With this parameter, the experimenter can force organisms to perform reactions in a specified order. Default: `reaction=none`

`no_reaction` Prevents the reaction from being triggered if the reaction specified here has already been triggered. This parameter enables the experimenter to define mutually exclusive reactions and force organisms to diversify. Default: `no_reaction=none`

`min_count` Requires that the task used to trigger the reaction be performed a certain number of times before the trigger will actually occur. This parameter (along with `max_count`) allows the experimenter to specify different reactions depending on the number of times an organism has performed a task. Default: `min_count=0`

`max_count` Places a cap on the number of times a task can be done and still trigger the reaction. It allows the experimenter to limit the number of times a reaction can be done, as well as (along with `min_count`) to specify different reactions depending on the number of times an organism has performed a task. Default: `max_count=infinite`

We can simulate the pre-environment system of older versions of Avida (in which no resources were present and task performance was rewarded with a fixed bonus) with an environment file including only lines like the following:

```
REACTION AND and process:type=mult:value=4.0 requisite:
max_count=1
REACTION EQU equ process:type=mult:value=32.0 requisite:
max_count=1
```

The documentation files shipped with Avida provide more detailed information on how to configure the environment file.

4.5 The Events File

The events file controls events that need to occur throughout the course of an experiment, such as the output of data to files, as well as disturbances to or modifications of the population (such as extinction events or changes to the mutation rate).

The events file consists of a list of events that will be triggered either once or periodically. The format for each line is:

type timing event arguments

The *type* determines what kind of timings the event will be based on. Timing can be immediate (`i`), based on update (`u`), or based on generation (`g`).

The *timing* should only be included for non-immediate events. If a single number is given for timing, the event occurs at that update (or generation). A second number can be included (separated by a colon, `:`) to indicate at what intervals the event should be repeated. And if a third number is listed (again, colon separated), it specifies the last time at which the event can occur. For example, the type and timing `u 100:100:5000` indicates that the event first occurs at update 100, and repeats every 100 updates thereafter, until update 5000 is reached. An entry `g 10:10` causes the event to be triggered every 10 generations for the entire run.

The *event* is the event that should be triggered, and the *arguments* define how exactly the event should work when it is triggered. Each event has its own arguments.

For example, to print out all average measurements collected every 100 updates, starting at update 100, we write:

```
u 100:100 print_average_data
```

The next example uses the `print_data` event:

```
g 10000:10:20000 print_data dom_info.dat \
```

```
update,dom_fitness,dom_depth,dom_sequence
```

This entry has the following effect: Between generations 10,000 and 20,000, append the specified information to the file "dom_info.dat" every ten generations. Specifically, the first column in the file contains the update number, the second column contains the fitness of the dominant genotype, followed by the depth in the phylogenetic tree of the dominant genotype, and the last column lists the dominant organism's genome sequence.

See Section A2.1 in Appendix 2 for a guide to the output events, and Section A2.2 for the events that will directly affect the population.

5 Experimental Techniques

Just as there are many ways of configuring Avida experiments, there are also many ways of studying the data that come out of them. In this section we will describe the standard forms of output data available at the end of an experiment, and the built-in techniques to work with those data.

5.1 Output Files

Avida output files must be scheduled to be written as part of the events file. Specific files have been designed for regularly output data. For example, the event `print_tasks_data` will output the current update as the first column, and then each subsequent column will display the number of organisms that are capable of performing the associated task. The top of each file contains a key to the specific values that the columns represent. Over the course of a single experiment, each time this event is run the associated data will be written to the end of the file. See Section A2.1 in Appendix 2 for a complete list of these periodic output events and a guide to the values in the columns of each file.

Other types of output can be triggered by events such as `print_dom`, which will print the dominant (most abundant) genotype to an output file in the `genebank/` directory. Two output events to take special notice of are `detail_pop` and `dump_historic_pop`, which will store information that Avida has saved about the genotypes in the population—the first of these events prints out all of the existing genotypes in the population, and the second prints out all ancestors of the current population. Both of these files have the same format, which includes all of the basic information about each genotype (one per line in the files). Note that in order to use `dump_historic_pop`, the parameter `TRACK_MAIN_LINEAGE` in the genesis file needs to be set to 1. The files created by `detail_pop` and `dump_historic_pop` can later be post-processed in the analyze mode (see next subsection). These files are useful because they represent the state of the entire population at the time point when they were created.

5.2 Test Environments

Often, when examining populations in Avida, the user will need to know the fitness or some other characteristic of an organism that has not yet gone through a full gestation cycle during the course of the run. For this reason, we have constructed a *test environment* in which organisms can run outside the population, and without affecting the population. The test environment will run the organism for at least one gestation cycle; it can be used either during a run or as part of the post-processing of Avida data. While an organism is run in a test environment, Avida keeps track of all the vital statistics of that organism, such as how many instructions the organism executes in one gestation cycle, which parts of its genome it executes or copies, which tasks it carries out, and so on.

When an organism is loaded into a test environment, its instructions are executed until it produces a viable offspring or a timeout is reached. In principle, we would want to be able to determine exactly whether an organism can replicate or not, but unfortunately, that is not possible. (Testing whether an organism can replicate amounts to solving the Halting Problem in computer science.) Therefore, at some point we must give up on any program we are testing and assume that it is not going to replicate. If age-based death is turned on in the population, then the maximum age organisms can reach in the population is a good limit on how long an organism in the test environment should be run.

One of the main goals of running an organism in a test environment is to determine whether it is viable and what its fitness is. However, for certain organisms it can be difficult to perform this test. We have already mentioned the halting problem. Another problem is caused by implicit mutations. For example, we might determine that an organism does produce an offspring, but that this offspring is not identical to its parent. In this case, if the offspring is not viable, then certainly the parent is also not viable. Therefore, we take the next step of continuing to run the offspring in the test environment, and if necessary its offspring, until we either find a self-replicator or a sustainable cycle. By default we will only test three levels of offspring before we assume the original organism to be nonviable. However, such cases happen very rarely, and not at all if implicit mutations are turned off in the genesis file.

Two final problems with the test environments are that they do not properly reflect the levels of limited resources (this can be difficult to know, particularly if we are post-processing) and that they do not handle any special interactions with other organisms, since only one is being tested at a time. Both of these issues are currently being examined, and we plan to have a much-improved test environment in the future. Test environments do, however, work remarkably well in most circumstances.

In addition to reconstructing statistics about organisms as they existed in the population, it is also possible to determine how an organism would have fared in an alternate environment, or even to construct entirely new genomes to determine how they perform. This last approach includes techniques such as performing all single-point mutations on a genome and testing each result to determine what its local fitness landscape looks like, or artificially crossing over pairs of organisms to determine their viability (as used for species determination, described in Section 2.5 above). Test environments are most commonly used in the post-processing of Avida data, as described in the next section.

5.3 Analyze Mode

Avida has an analysis-only mode (briefly, the *analyze mode*), which allows for powerful post-processing of data. Avida is brought into the analyze mode by the command-line parameter `-a`. In analyze mode, Avida processes the analyze file specified in the genesis file ("analyze.cfg" by default). The analyze file contains a program written in a simple scripting language. The structure of the program involves loading in genotypes in one or more *batches*, and then either manipulating single batches, or doing comparisons between batches.

In the following paragraphs, we present several example programs that will illustrate the basics of the analyze scripting language. A full list of commands available in analysis mode is given in Appendix 3.

5.3.1 Testing a Genome Sequence

The following program will load in a genome sequence, run it in a test environment, and output the results of the tests in a couple of formats:

```
VERBOSE
LOAD_SEQUENCE rmzavcgmciqqptqpqcpctletncogcbeamqdtqcptipqfpgqxuty
  cuastttva
RECALCULATE
DETAIL detail_test.dat fitness merit gest_time length viable
  sequence
TRACE
PRINT
```

The program starts off with the VERBOSE command, which causes Avida to print to screen all details of what is going on during the execution of the analyze script; the command is useful for debugging purposes. The program then uses the LOAD_SEQUENCE command to define a specific genome sequence in compressed format. (The compressed format is used by Avida in a number of output files. The mapping from instructions to letters is determined by the instruction set file, and may change if the instruction set file is altered.)

The RECALCULATE command places the genome sequence into the test environment, and determines the organism's fitness, merit, gestation time, and so on. The DETAIL command that follows prints this information into the file "detail_test.dat". (This filename is specified as the first argument of DETAIL). The TRACE and PRINT commands will then print individual files with data on this genome, the first tracing the genome's execution line by line, and the second summarizing several test results and printing the genome line by line. Since no directory was specified for these commands, "genebank/" is assumed, and the filenames are "org-S1.trace" and "org-S1.gen". If a genotype has a name when it is loaded, then that name will be kept. Otherwise, it will be assigned a name starting with "org-S1", then "org-S2", and so on. The TRACE and PRINT commands add their own suffixes (".trace" and ".gen") to the genome's name to determine the filenames they will use.

5.3.2 Using Variables

Often, it is necessary to run the same section of analyze code on multiple data sets, or it might be useful to be able to easily change settings throughout an analyze script. To facilitate such programming practices, variables are available in analyze mode.

There are actually several types of variables, all of which are denoted by a single letter or number. For a command that requires a variable name as an input, we give the name of that variable where it is requested. For example, in order to set the variable i to the value 12, we have to type

```
SET i 12
```

But later on in the code, how does Avida know whether an i is meant to be the letter or the value of the variable? To distinguish these cases, variables are marked by a leading dollar sign ($) wherever they are meant to be translated into their value.

There are several commands with which variable values can be manipulated. Some of these commands execute a section of code multiple times, each time with a different value for the variable. Here is one example:

```
FORRANGE i 100 199
  SET d /home/charles/dev/Avida/runs/evo-neut/evo_neut_$i
  PURGE_BATCH
  LOAD_DETAIL_DUMP $d/detail_pop.100000
  RECALCULATE
```

```
  DETAIL $d/detail.dat update length fitness sequence
END
```

The FORRANGE command runs the contents of the loop once for each possible value in the range, setting the variable i to each of these values in turn. Thus the first time through the loop, i will be equal to 100, then 101, 102, and so on, all the way up to 199. In this particular case, we have 100 runs (numbered 100 through 199) that we want to work with.

The first thing we do once we are inside the loop is to set the value of the variable d to the name of the directory we are going to be working with. Since this directory name is long, we do not want to have to type it every time we need it. If we set the variable d to it, then all we need to do thereafter is to type $d. Note that in this case we are setting a variable to a string instead of a number; that is fine, and Avida will figure out how to handle the contents of the variable properly. The directory we are working with changes each time the loop is executed, since the variable i is part of the directory name.

We then use the command PURGE_BATCH to get rid of all genotypes from the last execution of the loop (so as not to accumulate more and more genotypes in the current batch), and refill the batch by using LOAD_DETAIL_DUMP to load in all genotypes saved in the file "detail_pop.100000" within our chosen directory. The RECALCULATE command runs all of the genotypes through a test environment, so that we have all the statistics we need, and finally DETAIL prints out the chosen statistics to the file "detail.dat", again placing it into the proper directory. The END command signifies the end of the FORRANGE loop.

5.3.3 Finding Lineages

The portion of an Avida run that we will often be most interested in is the lineage from a genotype (typically the final dominant genotype) back to the original ancestor. There are tools in the analyze mode to obtain this information, if the necessary population and ancestral dumps have been written out with the events detail_pop and dump_historic_pop. The following program demonstrates how to make use of these dump files:

```
FORRANGE i 100 199
  SET d /home/charles/dev/Avida/runs/evo-neut/evo_neut_$i
  PURGE_BATCH
  LOAD_DETAIL_DUMP $d/detail_pop.100000
  LOAD_DETAIL_DUMP $d/historic_dump.100000
  FIND_LINEAGE num_cpus
  RECALCULATE
  DETAIL lineage.$i.html depth parent_dist length fitness
    html.sequence
END
```

The program looks similar to the previous one. The first four lines are actually identical, but after loading the detail dump at update 100,000, we also load the history dump from the same time point. A detail file contains all of the genotypes that were currently alive in the population at the time the detail file was printed, while a history file contains all of the genotypes that are ancestors of those that are still alive. The combination of these two files gives us the lineages of the entire population back to the original ancestor. Since we are only interested in a single lineage, we next run the FIND_LINEAGE command to pick out a single genotype, and discard everything else

except for its lineage. In this case, we pick the genotype with the highest abundance (i.e., the largest number of organisms, or virtual CPUs, associated with it) at the time of output.

As before, the RECALCULATE command gets us any additional information we may need about the genotypes, and then we print that information to a file using the DETAIL command. The filenames that we are using this time have the format "lineage.$i.html", that is, the files are all written to the current directory, with filenames that incorporate the run number. Also, because the filename ends in the suffix ".html", Avida prints the file in HTML format, rather than in plain text. Note that the specific values that we choose to print take advantage of the fact that we have a lineage (and hence have measured things like the genetic distance to the parent) and are in HTML mode (and thus can print the sequence using colors to specify where exactly mutations occurred).

6 Outlook

Digital organisms are a powerful research tool that has opened up methods to experimentally study evolution in ways never before possible. We have explained the capabilities of the Avida system, and detailed the methods by which researchers can make use of them. One must be careful, however, not to be lured into the trap of thinking that, because these systems can be set up and examined so easily, any experiment will be possible. There are definite limits on the questions that can be answered.

Using digital organisms, we cannot learn anything about physical structures evolved in the natural world, nor the specifics of an evolutionary event in our own history; the questions we ask must be about *how* evolution works in general, and how we can harness it. Even for the latter type of question, it is sometimes difficult to set up experiments in such a way that they give meaningful results. We must always remember that we are working with an arguably living system that will evolve to survive as best it can, not always in the direction that we intend. For example, in one experiment we wanted to study a population that could not adapt, but that would nevertheless accumulate deleterious or neutral mutations through drift. We thought that this situation was perfect to study with digital organisms: With the fine control over the system inherent in digital life, we could examine each mutation as it occurred by running a copy of the mutated organism in a test environment and measuring its fitness. We then killed those organisms in the main population for which the test revealed that they had gotten a beneficial mutation, thereby in theory stopping all future adaptation. We were shocked, however, to see that the population continued to evolve, albeit with a slow start, but ramping up to the normal evolutionary rates later on. Upon further investigation, it turned out that the organisms developed a method of detecting the fixed inputs that we provided in the test environments—and once they determined that they were in a test environment, they purposefully downgraded their own performance so as not to be killed. As a colleague put it, "they evolved predator avoidance." Remarkably, even when we removed all differences between the test environment and the real environment, we still could not prevent adaptation. The organisms then shifted to probabilistically expressing their complex features. With probabilistic expression, they had a chance of appearing to have low fitness in the test environment, even though their actual fitness was quite high. While some people initially saw these results as a failure in the Avida system, it is unexpected evolution such as this one that has convinced many biologists that digital organisms are all the more lifelike.

The genesis file still contains the option STERILIZE_BENEFICIAL (which is meant to disable further adaptation), but this option now compares the realized fitness of both the parent and offspring within the population. The parent's fitness is saved; the offspring then has to live out its entire life, and only when it, in turn, is about to

give birth can we be sure if it is more fit than its parent. At that point we cause the replication to fail. This method is further away from the occurrence of the mutation than we would like, but since it focuses on events in the population itself, it is the only way to halt adaptation.

The quest to halt adaptation is only one example of a special feature in Avida; many more have been explored, and are continuously being added to the source code. The most successful features are all fully described in the documentation that comes with the software.

From this point, many possible future directions exist for the development of Avida. Ongoing efforts include (among others) the implementation of a new CPU model that is more powerful and realistic, an overhaul of the graphical user interface so that it can easily be used by those not familiar with the software, an expanded analyze mode based on the scripting language Python, and the move from asexual to sexual organisms. We hope for these additions to expand the user base of the software as well as the range of experiments possible.

Acknowledgments
This work was supported by NSF grant DEB-9981397.

References

1. Adami, C., Brown, C. T., & Haggerty, M. R. (1995). Abundance distributions in artificial life and stochastic models: "Age and area" revisited. *Lecture Notes in Artificial Intelligence*, *929*, 503–514.

2. Adami, C., Ofria, C., & Collier, T. C. (2000). Evolution of biological complexity. *Proceedings of the National Academy of Sciences of the U.S.A.*, *97*, 4463–4468.

3. Barton, N., & Zuidema, W. (2003). Evolution: The erratic path towards complexity. *Current Biology*, *13*, R649–R651.

4. Chu, J., & Adami, C. (1997). Propagation of information in populations of self-replicating code. In C. G. Langton & T. Shimohara (Eds.), *Proceedings of Artificial Life V* (pp. 462–469). Cambridge, MA: MIT Press.

5. Cooper, T., & Ofria, C. (2002). Evolution of stable ecosystems in populations of digital organisms. In R. K. Standish, M. A. Bedau, & H. A. Abbass (Eds.), *Proceedings of Artificial Life VIII* (pp. 227–232). Cambridge, MA: MIT Press.

6. Darwin, C. (1859). *On the origin of species by means of natural selection.* London: Murray.

7. Dennett, D. (2002). The new replicators. In M. Pagel (Ed.), *Encyclopedia of evolution* (pp. E83–E92). Oxford, UK: Oxford University Press.

8. Dewdney, A. K. (1984). In a game called core war hostile programs engage in a battle of bits. *Scientific American, 250*(4), 14–22.

9. Egri-Nagy, A., & Nehaniv, C. L. (2003). Evolvability of the genotype-phenotype relation in populations of self-replicating digital organisms in a Tierra-like system. *Lecture Notes in Artificial Intelligence*, *2801*, 238–247.

10. Elena, S. F., & Lenski, R. E. (2003). Evolution experiments with microorganisms: The dynamics and genetic bases of adaptation. *Nature Reviews Genetics*, *4*, 457–469.

11. Kim, Y., & Stephan, W. (2003). Selective sweeps in the presence of interference among partially linked loci. *Genetics*, *164*, 389–398.

12. Lenski, R. E., Ofria, C., Pennock, R. T., & Adami, C. (2003). The evolutionary origin of complex features. *Nature*, *423*, 129–144.

13. McVean, G. A. T., & Charlesworth, B. (2000). The effects of Hill-Robertson interference between weakly selected mutations on patterns of molecular evolution and variation. *Genetics*, *155*, 929–944.

14. O'Neill, B. (2003). Digital evolution. *PLoS Biology*, *1*, 011–014.

15. Orr, H. A. (2000). The rate of adaptation in asexuals. *Genetics*, *155*, 961–968.

16. Rasmussen, S., Knudsen, C., Feldberg, R., & Hindsholm, M. (1990). The coreworld—Emergence and evolution of cooperative structures in a computational chemistry. *Physica D*, *75*, 1–3.

17. Ray, T. S. (1992). An approach to the synthesis of life. In C. G. Langton, C. Taylor, J. D. Farmer, & S. Rasmussen (Eds.), *Proceedings of Artificial Life II* (p. 371). Reading, MA: Addison-Wesley.

18. Travisano, M., & Rainey, P. B. (2000). Studies of adaptive radiation using model microbial systems. *American Naturalist*, *156*, S35–S44.

19. Wilke, C. O. (2002). Maternal effects in molecular evolution. *Physical Review Letters*, *88*, 078101.

20. Wilke, C. O., & Adami, C. (2002). The biology of digital organisms. *Trends in Ecology & Evolution*, *17*, 528–532.

21. Wilke, C. O., Wang, J. L., Ofria, C., Lenski, R. E., & Adami, C. (2001). Evolution of digital organisms at high mutation rates leads to survival of the flattest. *Nature*, *412*, 331–333.

22. Yedid, G., & Bell, G. (2001). Microevolution in an electronic microcosm. *American Naturalist*, *157*, 465–487.

23. Yedid, G., & Bell, G. (2002). Macroevolution simulated with autonomously replicating computer programs. *Nature*, *420*, 810–812.

Appendix 1 Instruction Sets

A1.1 Complements and Nop Modification

Nop commands are complemented as follows (here A $\longrightarrow$ B means B is the complement of A):

`nop-A` $\longrightarrow$ `nop-B`
`nop-B` $\longrightarrow$ `nop-C`
`nop-C` $\longrightarrow$ `nop-A`

Registers are complemented as follows:

`AX` $\longrightarrow$ `BX`
`BX` $\longrightarrow$ `CX`
`CX` $\longrightarrow$ `AX`

Nop modification for registers works as follows (here A $\leadsto$ B means instruction A indicates register B):

`nop-A` $\leadsto$ `AX`
`nop-B` $\leadsto$ `BX`
`nop-C` $\leadsto$ `CX`

Nop modification for heads works as follows:

`nop-A` $\leadsto$ instruction head
`nop-B` $\leadsto$ read head
`nop-C` $\leadsto$ write head

A1.2 The Default Instruction Set

Below, we explain the instructions of the default instruction set. We use the notation `?register?` to indicate that an instruction acts by default on the given register, but that a nop instruction following the instruction changes the default register according to the rules of nop modification. We use the same notation for nop modification on heads.

`nop-A` Do nothing.

`nop-B` Do nothing.

`nop-C` Do nothing.

`if-n-equ` If the `?BX?` register does not equal its complement register, execute the next instruction; otherwise skip it. (Thus `nop-A` following this command causes `AX` and `BX` to be compared; `nop-B`, the default, compares `BX` and `CX`, and finally, `nop-C` compares `CX` and `AX`.)

`if-less` If the `?BX?` register is smaller than its complement register, execute the next instruction; otherwise skip it.

`swap` Swap the contents of the `?BX?` register and its complement register.

`pop` Pop the top value on the currently active stack into register `?BX?`.

`push` Push register `?BX?` onto the currently active stack.

`swap-stk` Swap the currently active stack (there are two stacks in total).

`shift-r` Shift the bits in the `?BX?` register to the right.

`shift-l` Shift the bits in the `?BX?` register to the left.

`inc` Increment the `?BX?` register.

`dec` Decrement the `?BX?` register.

`add` Set `?BX?` equal to the sum of the `BX` and `CX` registers: `?BX? = BX + CX`.

`sub` `?BX? = BX - CX`.

`nand` `?BX? = BX` NAND `CX` (bitwise).

`h-alloc` Allocate the maximum number of instructions that a child organism may have on divide.

`h-divide` Split off the instructions between the read head and the write head, and turn them into a new organism. If there are instructions between the write head and the end of the memory, discard these. There are a number of conditions under which a divide will fail. Those are:

1. If either the parent or the offspring would have less than 10 instructions.
2. If the parent has not allocated memory.
3. If less than 70% of the parent was executed.
4. If less than 70% of the offspring's memory was copied into.
5. If the offspring would be too small or too large (as defined by the experimenter).

`IO` Do a `put` and a `get` immediately one after the other. Working register is `?BX?`.

`h-copy` Copy an instruction from the read head to the write head (possibly doing a mutation), and advance both heads.

`h-search` Search in the forward direction for the complement label, and set the flow control head to the end of the label. The distance to the end of the label found is placed in the `BX` register, and the size of the label in `CX`. If a complement label is not found or no label follows the instruction, the flow control head is set to the current position of the instruction head.

`mov-head` Move the `?`instruction head`?` to the position of the flow control head.

`jmp-head` Advance the `?`instruction head`?` by `CX` positions, and set the `CX` register to the initial position of the head.

`get-head` Write the position of the `?`instruction head`?` into the `CX` register.

`set-flow` Set the `?`flow control head`?` to the address pointed to by the `?CX?` register.

`if-label` If the label after this instruction is the complement of the most recently copied instructions, execute the next instruction after the label; otherwise skip it.

A1.3 Simple Organism for the Default Instruction Set

The following is a simple self-replicating program written in the default instruction set. This program is used as the default starting organism in Avida.

01	`h-alloc`	Allocate space for child
02	`h-search`	Locate the end of the organism
03	`nop-C`	Label α
04	`nop-A`	
05	`mov-head`	Place write head at the beginning of the offspring
06	`nop-C`	Nop modifier for `mov-head`
07	`h-search`	Mark the beginning of the copy loop
08	`h-copy`	Do the copy
09	`if-label`	If we're done copying...
10	`nop-C`	Label α
11	`nop-A`	
12	`h-divide`	...divide!
13	`mov-head`	Otherwise, loop back to the beginning of the copy loop
14	`nop-A`	Label $\bar{\alpha}$
15	`nop-B`	

A1.4 The Old Instruction Set (without Heads)

The following instructions are identical to the ones from the default set: `nop-A`, `nop-B`, `nop-C`, `if-n-equ`, `if-less`, `pop`, `push`, `swap-stk`, `swap`, `shift-r`, `shift-l`, `inc`, `dec`, `add`, `sub`, `nand`. Below, we explain the instructions that differ from the default instruction set.

`if-bit-1` If bit 1 of the `?BX?` register is set, execute the next instruction; otherwise skip it.

`jump-f` If a label follows, search for its complement in the forward direction; if a match is found, jump to it. If there is no label, jump by `BX` instructions in the proper direction. If there is a label, but its complement is not found, do nothing.

`jump-b` Same as `jump-f`, but in backward direction.

`call` Put the location of the next instruction on the currently active stack (there are two stacks in total), and jump forward to the complement of the label that follows. If there is no label, jump `BX` instructions.

`return` Pop the top value from the currently active stack, and jump to that index in the organism's memory.

`copy` Copy a command from the memory location pointed to by the `BX` register to the memory location pointed to by `AX + BX`, that is, copy the instruction at location `BX` into a location *offset* by `AX`. If a location is out of range of the memory, it will be cycled back into range. The copy process is not error free, which means that sometimes a random instruction is written instead of the instruction at `BX`. The rate at which this happens is the copy mutation rate, controlled in the genesis file.

`allocate` Allocate memory for `?BX?` instructions at the end of the organism's memory, and return the start location of the new memory in `AX`. Only one allocate may occur between successful divides; any additional attempts to allocate will fail. Additionally, not more than double or less than half of the current memory size can be successfully allocated.

`divide` Split the memory of the organism at `AX`, turning the instructions beyond the divide point into a new organism. The same restrictions that have to be satisfied for `h-divide` have to be satisfied for `divide`.

`get` Read the next value from the input buffer into `?CX?`.

`put` Place `?BX?` in the output buffer, and set the register used to 0.

`search-f` Search in the forward direction for the complement label, and return its distance. The returned value is placed in the `BX` register, and the size of the label that followed is placed in `CX`. If a complement label is not found, a distance of 0 is returned.

`search-b` As `search-f`, but in the backward direction.

A1.5 Simple Organism for the Old Instruction Set

The following is a simple self-replicating program written in the old instruction set:

01	`search-f`	Find distance to the end label
02	`nop-A`	Label α
03	`nop-A`	
04	`add`	Account for label α's size
05	`inc`	Account for the initial `search-f`
06	`allocate`	Allocate space for the daughter
07	`push`	Push size from `BX` onto the stack
08	`nop-B`	Nop modifier for `push`
09	`pop`	Pop size off of the stack into `CX`
10	`nop-C`	Nop modifier for `pop`
11	`pop`	Since the stack is empty, pop 0 into `BX`
12	`nop-B`	Label $\bar{\beta}$ (copy loop start)
13	`nop-C`	
14	`copy`	Copy the current line
15	`inc`	Move on to the next line

16	`if-n-equ`	If we aren't done copying…
17	`jump-b`	…jump back to the loop's beginning
18	`nop-A`	Label β
19	`nop-B`	
20	`divide`	Done copying; separate the daughter!
21	`nop-B`	Label $\bar{\alpha}$
22	`nop-B`	

Appendix 2 Events

In the following sections, we describe the most commonly used events in Avida. As a formatting guide, command arguments will be presented between brackets, such as {`filename`}. Optional arguments will have a default value listed, such as {`filename= ``output.dat''`}.

A2.1 Output Events

Output events are the primary way of saving data from an Avida experiment. The main two types are continuous output, which appends to a single file every time the event is triggered, and singular output, which produces a single, complete file for each trigger.

`print_data` {`filename`} {`column_list`} Append the data given in the column list to the file specified (continuous output). The column list needs to be a comma-separated list of keywords representing the data types. Many possible data types can be output; see Section A2.3 for the complete list. The event will create a detailed column legend at the top of the file, so that the file format is always properly documented.

`print_average_data` A `print_data` shortcut that will append all of the population averages to the file "average.dat".

`print_error_data` A `print_data` shortcut that will append all of the standard errors of the population statistics to the file "error.dat".

`print_variance_data` A `print_data` shortcut that will append all of the variances of the population statistics to the file "variance.dat".

`print_dominant_data` A `print_data` shortcut that will append all of the statistics relating to the dominant genotype to the file "dominant.dat".

`print_stats_data` A `print_data` shortcut that will append all of the miscellaneous population statistics to the file "stats.dat".

`print_counts_data` A `print_data` shortcut that will append all of the statistics that keep track of counts (such as the number of organisms in the population or the number of instructions executed) to the file "count.dat".

`print_totals_data` A `print_data` shortcut that will append the same information as the previous event, but the counts will be the totals for the entire length of the run (for example, the total number of organisms ever) to the file "totals.dat".

`print_time_data` A `print_data` shortcut that will append all of the timing related statistics to the file "time.dat".

`print_tasks_data` Append the number of organisms that are able to perform each task to the file "tasks.dat". This event uses the environment configuration to

determine what tasks are in use.

`print_resource_data` Append the current counts of each resource available to the population to the file "resource.dat". This event uses the environment configuration to determine what resources are in use.

`detail_pop {filename}` Save the genotypes and additional data from the population to the file specified; if no file name is given, the event uses the name "detail_pop.*update#*", where "*update#*" is the update number at which the event is triggered. Columns in the output are as follows (one line per genotype): (1) genotype ID, (2) parent ID, (3) distance from parent, (4) current number of organisms of this genotype, (5) total number of organisms of this genotype, (6) length, (7) merit, (8) gestation time, (9) fitness, (10) update born, (11) update deactivated, (12) depth, (13) genome (in compressed format).

`dump_historic_pop {filename}` This event is used to output all of the ancestors of the currently living population to the file specified, or "historic_dump.*update#*". It uses the same format as the `detail_pop` event. In order to use this event, the parameter `TRACK_MAIN_LINEAGE` in the genesis file has to be set to 1.

A2.2 Population Events

Population events modify the state of the population, and will actually change the course of the run.

`inject {filename} {cell_id=0} {merit=-1} {lineage_id=0}` Inject a single organism into the population. Arguments must be included from left to right; if all arguments are left out, the organism chosen by default is the ancestral organism; it will be injected into cell 0, have its merit initialized to its genome length, and be marked as lineage 0.

`inject_all {filename} {merit=-1} {lineage_id=0}` Same as `inject`, but no `cell_id` is specified, and the organism is placed in all cells in the population.

`apocalypse {kill_prob}` Organisms in the population are killed randomly with the given probability.

`serial_transfer {num_organisms}` Similar to apocalypse, but here we can specify the exact number of organisms to keep alive after the event.

`kill_rectangle {X1} {Y1} {X2} {Y2}` Kill off all organisms in a rectangle defined by the points (`X1, Y1`) and (`X2, Y2`).

A2.3 Data Types

These are the data types available to the `print_data` command.

`update`	Current update
`sub_update`	Instructions executed within update
`generation`	Average generation in population
`entropy`	Genotype entropy (diversity)
`species_entropy`	Species entropy (diversity)
`energy`	Average inferiority (energy)

dom_merit	Average merit of dominant genotype
dom_gest	Average gestation time of dominant genotype
dom_fitness	Average fitness of dominant genotype
dom_repro	Average reproduction rate (1/gestation) of dominant genotype
dom_length	Genome length of dominant genotype
dom_copy_length	Copied length of dominant genotype
dom_exe_length	Executed length of dominant genotype
dom_id	ID of dominant genotype
dom_name	Name of dominant genotype
dom_births	Birth count of dominant genotype
dom_breed_true	Breed-true count of dominant genotype
dom_breed_in	Breed-in count of dominant genotype
dom_breed_out	Breed-out count of dominant genotype
dom_num_cpus	Abundance of dominant genotype
dom_depth	Tree depth of dominant genotype
dom_sequence	Sequence of dominant genotype
num_births	Count of births in population
num_deaths	Count of deaths in population
breed_in	Count of non-breed-true births
breed_true	Count of breed-true births
bred_true	Count of organisms that have bred true
num_cpus	Count of organisms in population
num_genotypes	Count of genotypes in population
num_threshold	Count of threshold genotypes
num_species	Count of species in population
thresh_species	Count of threshold species
num_lineages	Count of lineages in population
num_parasites	Count of parasites in population
num_no_birth	Count of childless organisms
tot_cpus	Total organisms ever in population
tot_genotypes	Total genotypes ever in population
tot_threshold	Total threshold genotypes ever
tot_species	Total species ever in population
tot_lineages	Total lineages ever in population

ave_repro_rate	Average reproduction rate
ave_merit	Average merit
ave_age	Average age
ave_memory	Average memory used
ave_neutral	Average neutral metric
ave_lineage	Average lineage label
ave_gest	Average gestation time
ave_fitness	Average fitness
ave_gen_age	Average genotype age
ave_length	Average genome length
ave_copy_length	Average copied length
ave_exe_length	Average executed length
ave_thresh_age	Average threshold genotype age
ave_species_age	Average species age
max_fitness	Maximum fitness in population
max_merit	Maximum merit in population

Appendix 3 Analyze Commands

This analysis language is a simple programming language. The structure of a program involves loading in genotypes into one or more *batches*, and then either manipulating single batches, or doing comparisons between batches. Currently there can be at most 300 batches of genotypes, but we will eventually remove this limit.

The rest of this section describes how individual commands work, and gives some notes on other language features, such as how to use variables. As a formatting guide, command arguments will be presented between brackets, such as {filename}. Optional arguments will have a default value listed, such as {filename=``output.dat''}.

A3.1 Genotype Loading Commands

There are currently four ways to load in genotypes:

LOAD_ORGANISM {filename} Load in a normal single-organism file of the type that is output from Avida from events such as print_dom. These files consist of information on the organism enclosed in comments, followed by the full genome of the organism one instruction per line.

LOAD_BASE_DUMP {filename} Load in a basic dump file from Avida. Each line contains a genotype sequence, but little additional information.

LOAD_DETAIL_DUMP {filename} Load in a detail file. Detail files are similar to the basic dump files, but contain a lot more information on each line. When saved from Avida (with events detail_pop or dump_historic_pop), these files typically begin with the word "detail" or "historic".

LOAD_SEQUENCE {sequence} Load in a user-provided sequence as the genotype. Avida has a symbol associated with each instruction; this command is followed by a sequence of such symbols that is then translated into a genotype.

A3.2 Batch Control Commands

All of the load commands place the new genotypes into the *current* batch, which can be set with the SET_BATCH command. The following are the commands that manipulate batches:

SET_BATCH {id} Set the batch that is currently active. The initial active batch at the start of a program is 0.

NAME_BATCH {name} Attach a name to the current batch. Some of the printing methods will print data from multiple batches, and we want the data from each batch to be attached to a meaningful identifier.

PURGE_BATCH {id=current} Remove all genotypes in the specified batch (if no argument is given, the current batch is purged).

DUPLICATE {id1} {id2=current} Copy the genotypes from batch id1 into id2. By default, copy id1 into the current batch. Note that DUPLICATE is nondestructive, which means that the genotypes that were previously in batch id2 remain there. In order to remove them, the command PURGE_BATCH has to be run on the batch id2 before the command DUPLICATE.

STATUS Print out (to the screen) the genotype count of each nonempty batch, and identify the currently active batch.

A3.3 More Analysis Control Commands

There are several other commands that allow the experimenter to interact with the analysis mode in important ways, but that do not actually trigger any analysis tests or output. Below is a list of some of the more important control commands:

VERBOSE Toggle between verbose and minimal messages. Verbose messages will print all of the details of what is happening to the screen. Minimal messages will only briefly state the process being run. Verbose messages are recommended in interactive mode.

SYSTEM {command} Run the given command on the underlying operating system. This command is useful to manipulate files, for example, to copy or delete them.

INCLUDE {filename} Include another file into this one, and run its contents immediately. This command is useful for prewritten routines that should be available in several analysis files. Warning: There are currently no protections against cyclical includes.

INTERACTIVE Switch the analyze mode into interactive. In interactive mode, it is possible to enter analyze commands on the command line, and these commands will be executed immediately. This command can be placed anywhere within the analyze file, so that some processing can take place before interactive mode starts. The command quit will switch back to batch mode, and Avida will continue with the normal processing of the analyze script.

DEBUG {message} This command will print the given message to the screen. If there are any variables (see Appendix A3.8) in the message, they will be translated before printing. Therefore, the command is useful for debugging programs.

A3.4 Genotype Manipulation Commands

Having explained how to interact with analysis mode and how to load in genotypes, we will now explain how genotypes can be manipulated. The following commands do basic analysis on genotypes, or can be used to prune batches to keep only those genotypes that meet specific criteria.

`RECALCULATE` Run all of the genotypes in the current batch through a test environment, and record the measurements taken (fitness, gestation time, and so on). This command overrides any values that may have been loaded in with the genotypes.

`FIND_GENOTYPE {type=``num_cpus'' ...}` Remove all genotypes apart from the ones selected. Type indicates which genotypes to choose. Options available for type are ``num_cpus'' (choose the genotype with the maximum abundance at time of printing), ``total_cpus'' (genotype with maximum number of organisms ever), ``fitness'' (genotype with highest fitness), and ``merit'' (genotype with highest merit). If the type entered is a number, it is used as an ID number to indicate the desired genotype (if no such ID exists, a warning will be printed to screen). Multiple arguments can be given to this command, in which case all those genotypes in the list will be preserved and the remainder deleted.

`FIND_LINEAGE {type=``num_cpus''}` Delete everything except the lineage from the chosen genotype back to the most distant ancestor available. This command will only function properly if parental information was loaded in with the genotypes. Type is the same as the `FIND` command.

`ALIGN` Create an alignment of all sequences in batch; gaps in the alignment will be indicated with _. Note that a `FIND_LINEAGE` must first be run on the batch; otherwise alignment is not possible.

`SAMPLE_ORGANISMS {fraction}` Keep only a fraction of organisms in the current batch. The sampling is done per organism, not per genotype. Thus, genotypes of high abundance may only have their abundance lowered, while genotypes of abundance 1 will either stay or be removed entirely.

`SAMPLE_GENOTYPES {fraction}` Keep only a fraction of genotypes in the current batch.

`RENAME {start_id=0}` Change the ID numbers of all genotypes in the batch to successive numbers starting at `start_id`. In long Avida experiments, ID numbers can grow to six or more digits. After reducing a batch to a lineage, in particular, it is useful to number the genotypes in order from the ancestor to the final one, rather than to keep the original six-digit numbers.

A3.5 Basic Output Commands

Next, we explain standard output commands that save information generated in analyze mode.

`PRINT {dir=``genebank/''}` Print the genotypes from the current batch as individual files (one genotype per file) in the directory given. The files will be named by the genotype name with ".gen" appended.

`TRACE {dir=``genebank/''}` Trace all of the genotypes and print a listing of their execution. The trace will show step by step the status of all of the CPU

components and the genome during the course of the execution. The filename used for each trace will be the genotype's name with ".trace" appended.

`PRINT_TASKS {file=``tasks.dat''}` This command will print out the tasks doable by each genotype, one per line, into the output file specified. Note that this information must either have been loaded in, or a `RECALCULATE` must have been run to collect it.

`DETAIL {file=``detail.dat''} {format ...}` Print out all statistics for each genotype, one per line. The format indicates the layout of columns in the file. If the filename specified ends in ".html", HTML formatting will be used instead of plain text. For the format, see Appendix A3.7.

A3.6 Analysis Commands

Finally, we list actual analysis commands that perform tests on the genomes and output the results.

`LANDSCAPE {file=``landscape.dat''} {dist=1}` For each genotype in the current batch, test all possible mutations (or combinations of mutations if `dist` > 1), and summarize the results, one per line in the specified file.

`MAP_TASKS {dir=``phenotype/''} {flags ...} {format ...}` Construct a genotype-phenotype array for each genotype in the current batch. The format is the list of data that should be included as columns in the array. Additionally, special format flags can be given; the possible flags are `html` to print output in HTML format, and `link_maps` to create html links between consecutive genotypes in a lineage.

`MAP_MUTATIONS {dir=``mutations/''} {flags ...}` Construct a genome-mutation array for each genotype in the current batch. Each line in the genome is given as a row in the chart, and all available instructions represent the columns. The cells in the chart indicate the fitness were a mutation to the listed instruction to occur at the position in the matrix. If the `html` flag is used, the charts will be output in HTML format.

`HAMMING {file=``hamming.dat''} {b1=current} {b2=b1}` Calculate the Hamming distance between batches `b1` and `b2`. If only one batch is given, calculations are done on all pairs within that batch.

`LEVENSTEIN {file=``lev.dat''} {b1=current} {b2=b1}` Calculate the Levenstein distance (edit distance) between batches `b1` and `b2`. This metric is similar to the Hamming distance, but calculates the minimized edit distance.

`SPECIES {file=``species.dat''} {b1=current} {b2=b1}` Again this command is similar to the Hamming distance, but calculates as if genotypes were considered the same species.

A3.7 Output Formats

Several commands (such as `DETAIL` and `MAP_TASKS`) require format parameters to specify what genotypic features should be output. Before such commands can be used, other collection functions may need to be run.

Allowable formats after a normal load (assuming these values were available from the input file that was loaded in) are:

`id` (genome ID)	`parent_id` (parent ID)
`num_cpus` (number of organisms)	`total_cpus` (total organisms ever)
`length` (genome length)	`update_born` (update born)
`update_dead` (update dead)	`depth` (tree depth)
`sequence` (genome sequence)	

After a `RECALCULATE`, several calculations are performed that make these additional formats available:

`viable` (is viable [0/1])	`copy_length` (copied length)
`exe_length` (executed length)	`merit` (merit)
`comp_merit` (computational merit)	`gest_time` (gestation time)
`efficiency` (replication efficiency)	`fitness` (fitness)

If a `FIND_LINEAGE` was done before the `RECALCULATE`, the parent genotype for each regular genotype will be available, enabling the additional formats:

`parent_dist` (parent distance)
`comp_merit_ratio` (ratio of `comp_merit` to parent's)
`efficiency_ratio` (ratio of efficiency to parent's)
`fitness_ratio` (ratio of fitness to parent's)
`parent_muts` (list of mutations from parent)
`html.sequence` (color genome sequence in HTML format)

Finally, if an `ALIGN` command is run, one additional format becomes available:

`alignment` (aligned sequence)

A3.8 Variables

Variables can only be a single character (letter or number) and begin with a `$` whenever they need to be translated to their value. Lowercase letters are global variables, capital letters are local to a function (described later), and numbers are arguments to a function. A `$$` will act as a single dollar sign, if needed.

`SET {variable} {value}` Set the variable to the value.

`FOREACH {variable} {value ...}` Set the variable to each of the values listed, and run the code that follows between the `FOREACH` command and the next `END` command once for each of those values.

`FORRANGE {variable} {min_value} {max_value} {step_value=1}` Set the variable to each of the values between `min` and `max` (in steps given), and run the code that follows between the `FORRANGE` command and the next `END` command, once for each of those values.

A3.9 Functions

Function support is currently primitive. Functions have fixed inputs of `$0` through `$9`. `$0` is always the function name; the remaining nine variables are arguments that can

be given to the function. Once a function is created, it can be run just like any other command.

`FUNCTION {name}` This command will create a function of the given name, including in it all of the commands up until the `END` command. These commands will be bound to the function, but are not executed until the function is run as a command. Inside the function, the variables `$1` through `$9` can be used to access arguments passed in.

Currently there are no conditionals or mathematical commands in this scripting language. These are both planned for the future.