DOI: http://dx.doi.org/10.7551/978-0-262-32621-6-ch114

Self-Replicating Distributed Virtual Machines

Lance R. Williams[1]

[1]Department of Computer Science, University of New Mexico, Albuquerque, NM 87131
williams@cs.unm.edu

Abstract

Recent work showed how an expression in a functional programming language can be compiled into a massively redundant asynchronous spatial computation called a *distributed virtual machine*. A DVM is comprised of bytecodes reified as actors undergoing diffusion and communicating via messages containing encapsulated virtual machine states. Significantly, it was shown that both the efficiency and the robustness of expression evaluation by DVM increase with redundancy. In the present work, spatial computations that become more efficient and robust over time are described. They accomplish this by self-replication, which increases the redundancy of the elements of which they are comprised. The first and simplest of these *self-replicating DVMs* copies itself by *reflection*; it reads itself from a contiguous range of memory. The remainder are *quines*. As such, they reproduce by translating and transcribing self-descriptions. The nature of the self-descriptions and of the translation and transcription processes differ in each case. The most complex self-replicating DVM described represents a fundamentally new kind of artificial organism–a machine language program reified as a spatial computation that reproduces by compiling its own source-code.

Introduction

In a recent article, Ackley (2013) argues that future robust computing systems will be defined in terms of *bespoke physics interfaces* built on top of a physical substrate resembling an *asynchronous cellular automaton (ACA)*; space and time in *indefinitely scalable* computations will be *coextensive* with physical space and time. von Neumann, who invented the *random access stored program computer (RASP)*, showed us that programs can be data. Yet conventionally, machines are active and data is passive; machines *transform* data. In this paper, we propose that computations formulated in terms of a bespoke physics are *literally* machines and demonstrate how self-replicating programs written in a Lisp-like language can be compiled into such computations.

A range of computational substrates have been used to host self-replicating programs in artificial life research or might play this role in the future. The *distributed virtual machine (DVM)* developed by the author as a method for the robust evaluation of expressions is in the second category (Williams, 2012). See Table 1.

Table 1: Computational substrates for artificial life.

	control	sites	bits/site	r/w dist.
CA	C	$N \times N$	O(1)	O(1)
ACA	D	$N \times N$	O(1)	O(1)
DVM	D	$N \times N \geq M$	$O(\log M)$	O(1)
Avida	C	$N \times N$	$O(P) + Q \log P$	P
Tierra	C	P	$O(1) + Q \log P$	P
RASP	C	1	$M \log M$	M

Notwithstanding their historical importance, CAs are likely to be supplanted by ACAs for two reasons. First, their dependence on a global clock violates Ackley's requirement that a bespoke physics be indefinitely scalable. Second, it is well known that for every CA there exists an ACA which emulates it (Nakamura, 1974; Nehaniv, 2004). Less well known is the fact that emulations of this type can be done with negligible slowdown (Berman and Simon, 1988).

Tierra (Ray, 1994) and Avida (Adami et al., 1994) exemplify the artificial life approach to evolutionary computation. Both systems are based on a *virtual machine (VM)* that has two distinct address spaces. The first (used to hold programs) consists of P words of size O(1) bits while the second (used to hold a stack of program memory addresses at runtime) consists of Q words of size $\log P$ bits. In contrast, the DVM and RASP both have a single address space consisting of M words of size $\log M$ bits; the major difference between the DVM and RASP is that the DVM's memory is spatially distributed over an $N \times N$ grid.

Moreover, like the RASP, the Tierra and Avida VMs are random access, with instructions that can read and write values anywhere in a memory of size P. In contrast, a DVM can be implemented on an ACA substrate with only $O(\log M)$ bits per site and with a read/write distance which is O(1).

Finally, the table reveals an important difference between Tierra and Avida, namely that all organisms occupy a single address space in Tierra but are segregated in separate address spaces in Avida. The difference is not minor–in Tierra the competition for program memory P is zero sum while in Avida it isn't. However, organisms which evolve more efficient methods of self-replication in terms of stack memory Q go unrewarded in *both* systems. In contrast, in a DVM, the competition for heap-allocated space M (no matter its use) among all organisms sharing the $N \times N$ grid is zero sum.

A Simple Programming Language

The language we used to construct our self-replicating DVMs is a pure functional subset of Scheme which we call *Skeme*. Because it is purely functional, *define*, which associates values with names in a global environment using mutation, and *letrec*, which also uses mutation, have been excluded. The global environment itself is eliminated by making primitive functions constants. For simplicity, closures are restricted to one argument; user defined functions with more than one argument must be written in a curried style. This simplifies the representation of the lexical environment which is used at runtime by making all variable references integer offsets into a flat environment stack; these are termed *de Bruijn indices* and can be used instead of symbols to represent bound variables (De Bruijn, 1972).

One feature peculiar to Skeme is the special-form, *lambda+*. When a closure is created by *lambda+*, the closure's address is added to the front of the enclosed environment; the de Bruijn index for this address can then be used for recursive function calls. For example, the following function computes factorial:

```
(lambda+ (if (= %0 0) 1 (* %0 (%1 (- %0 1)))))
```

where %0 is a reference to the closure's argument and %1 is a reference to the closure's address.

Evaluation of Expressions by Virtual Machines

The process of evaluating expressions by compiling them into bytecodes which are executed on a VM was first described by Landin (1964) for Lisp and was generalized for Scheme by Dybvig (1987). Because it plays an important role in our work, it is worth examining Dybvig's model for Scheme evaluation in some detail.

Evaluating an expression requires saving the current evaluation context onto a stack, then recursively evaluating subexpressions and pushing the resulting values onto a second stack. The second stack is then reduced by applying either a primitive function or a closure to the values it contains. Afterwards, the first stack is popped, restoring the prior evaluation context. Expressions are compiled into trees of bytecodes which perform these operations when the bytecodes are interpreted. For book keeping during this process, Dybvig's VM requires five registers. See Figure 1.

With the exception of the *accumulator*, which can point to an expression of any type, and the *program counter*, which points to a position in the tree of bytecodes, each of the registers in the VM points to a heap allocated data structure comprised of pairs; the *environment* register points to a stack representing definitions in enclosing lexical scopes, the *arguments* register points to the stack of values which a function (or closure) is applied to, and the *frames* register points to a stack of suspended evaluation contexts.

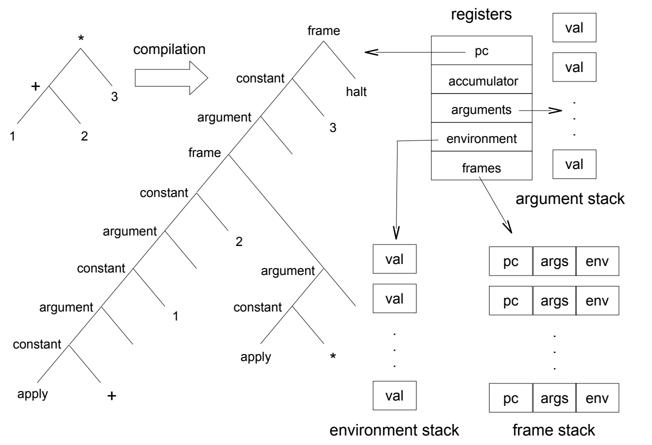

Figure 1: Virtual machine for evaluating compiled Skeme expressions showing its registers and associated heap-allocated data structures (based on Dybvig (1987)).

Evaluation occurs as the contents of these registers are transformed by the interpretation of the bytecodes. For example, the *argument* bytecode pushes the value of an evaluated subexpression onto the argument stack. Other bytecodes alter the frame stack. For example, the *frame* bytecode pushes an evaluation context onto the frame stack, while the *apply* bytecodes pops it after applying a primitive function (or a closure) to the values found in the argument stack. Still other bytecodes load the accumulator. For example, the *constant* bytecode loads it with a constant, while the *refer* bytecode loads it with a value found in the environment stack.

Reified Actor Models

Actors are universal primitives for constructing concurrent computations introduced by Hewitt et al. (1973). In essence, an actor is a lightweight process with a unique address which can send and receive messages to and from other actors. In response to receiving a message, and (depending on the message's contents) an actor can send a finite number of messages of its own; create a finite number of new actors; and change its internal state so that its future behavior is different. All of these things happen asynchronously.

Although as originally conceived, actor models are not reified, it is possible to create a *reified actor model*. In a reified actor model, all actors have unique positions on a 2D grid. Actors possess a finite number of states and can sense and change the positions and states of actors in their $n \times n$ *neighborhoods.* Significantly, actors can create *bonds* with other actors; bonds are *relative spatial addresses* which are short, symmetric, and automatically updated as actors undergo random independent motion or *diffusion.* Since bonds are short, they restrict the motion of an actor which possesses them. However, they also ensure that two actors which must communicate can always do so.

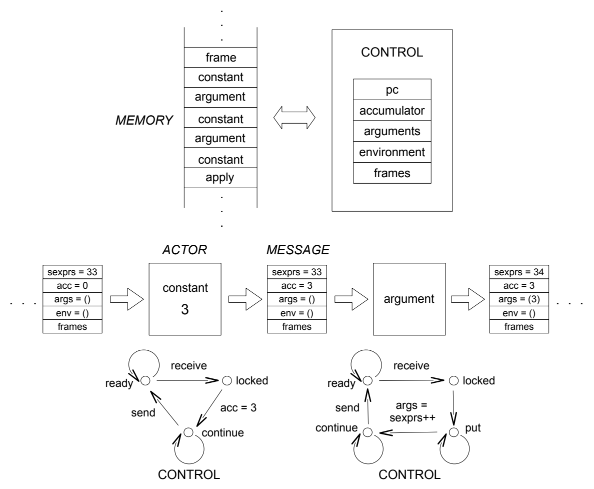

Figure 2: Conventional *virtual machine* (top) and *distributed virtual machine* (bottom). In the DVM, the registers are encapsulated in a message called a *continuation* that is passed between bytecodes reified as actors. Each actor is a finite state machine that transforms the continuation in manner specific to its type then passes it to the next bytecode in the program. Control is distributed not centralized.

Distributed Virtual Machines

By giving them addresses and types, reified actors can be used to represent heap-allocated objects. In particular, they can be used to represent any of the datatypes permissible in Skeme including numbers, booleans, primitive functions, de Bruijn indices, closures and pairs. However, they can also represent the bytecodes of a compiled Skeme program. We call the set of *bytecode actors* representing a compiled program, a *distributed virtual machine (DVM)*. Like other objects, a bytecode actor will respond to a *get* message by returning its value, but unlike actors representing other objects, it can also send and receive encapsulated virtual machine states, or *continuations*. Upon receipt of a continuation, a bytecode actor transforms it in a manner specific to its type, then passes it on to the next bytecode actor in the program, and so on, until the continuation reaches a *halt* bytecode. In contrast to a conventional VM, where all control is centralized, control in a DVM is distributed among the bytecodes which comprise it. Furthermore, because Skeme is purely functional, multiple instances of each object and multiple execution threads (continuations) can coexist without inconsistency in the same heap.

Recall that applying a function requires the construction of a stack of evaluated subexpressions. In the simplest case, these subexpressions are constants, and the stack is constructed by executing the *constant* and *argument* bytecodes in alternation. This two bytecode sequence is used to illustrate the operation of a DVM in more detail.

A bytecode actor of type *constant* in the *locked* state loads its accumulator with the address of its operand and enters the

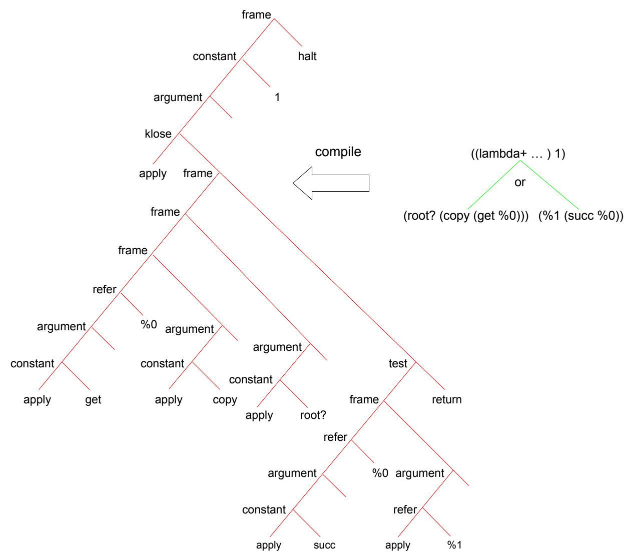

Figure 3: A self-replicating object code program consisting of 37 bytecodes and operands that copies itself by reflection (left) and its Skeme source code (right).

continue state. When an actor in the *continue* state sees its child in the bytecode tree within its neighborhood, it overwrites the child actor's registers with the contents of its own registers, sets the child actor's state to *locked*, and returns to the *ready* state.

The behavior of a bytecode actor of type *argument* in the *locked* state is more complicated. It must push its accumulator onto the argument stack, which is comprised of heap-allocated pairs. Since this requires allocating a new pair, it remains in the *put* state until it sees an adjacent empty site in its neighborhood. After creating the new pair actor on the empty site, it increments the register representing the last allocated heap address (for the execution thread) and enters the *continue* state. See Figure 2.

Self-Replication by Reflection

Since machine language programs are just sequences of values stored in memory, and since instruction sets include instructions which read from and write to memory, it is straightforward to construct a machine language program which copies itself using *reflection.* Primitive functions providing comparable functionality can be added to Skeme. The most important of these are: *copy* which makes a copy of a heap-allocated object; *root?* which returns true if its argument is the last heap-allocated object in an object code program and false otherwise; and *get*, which given an integer address, returns the heap-allocated object with that address. For convenience, we also add a primitive function *succ* which returns the successor of its integer argument. Using these functions, it is possible to define a Skeme expression which will copy a set of heap-allocated objects representing the bytecodes and operands of an object code pro-

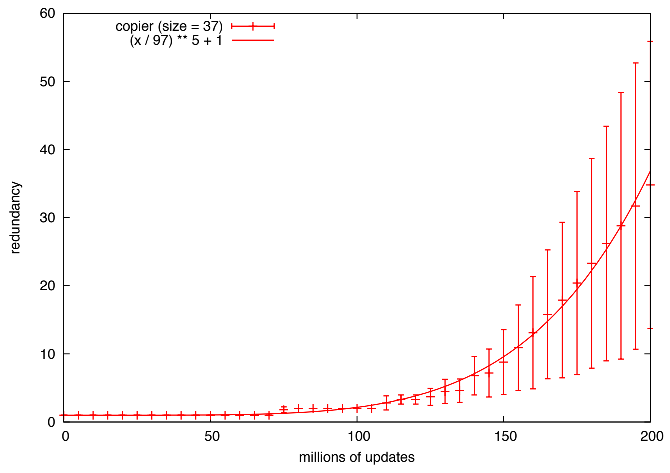

Figure 4: Redundancy versus time for reflection-based SRDVM using diffusion for message passing and best fit function of the form $f(x) = (x/a)^b + 1$. The average is over ten runs. Grid size was 256×256. Error bars show $\pm\sigma$. The computation becomes more efficient and robust over time.

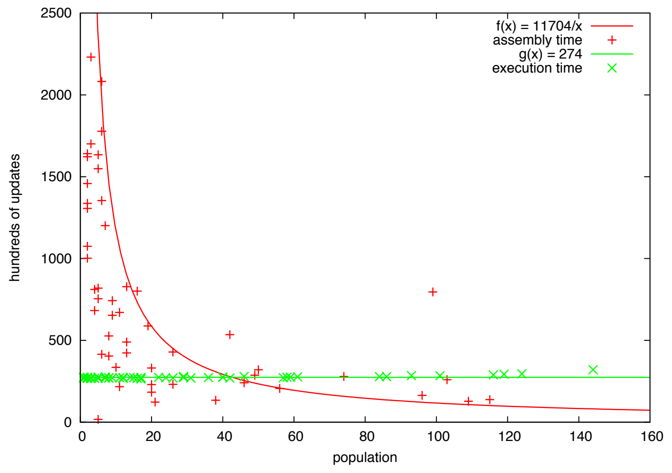

Figure 5: Times required for self-assembly (red) and execution (green) for reflection-based SRDVM as a function of population and best fit functions of the forms $f(x) = a/x$ and $g(x) = b$. The plotted times are for a single frame bytecode accumulated over ten runs. Grid size was 128×128. Self-assembly time decreases with increasing population while execution time remains constant.

gram stored in a contiguous range of addresses. This expression compiles into an object code program containing 37 bytecodes and operands (Figure 3). When reified as actors undergoing diffusion, the bytecodes and operands of the compiled program form a *self-replicating DVM*.

Execution begins when the *root actor* (the actor with largest address among the bytecodes which comprise the program) is sent an initial continuation. When execution ends, two new continuations are launched. To accomplish this, a behavior was added to the actor representing the *copy* primitive function. When the root actor is copied (the penultimate step in execution), both the root and the copy are sent initial continuations. Because the parent continuation dies a short time later (when it reaches the *halt* bytecode), the net increase in continuations is one. This ensures that the number of execution threads equals the redundancy increase due to self-replication. However, self-replication alone cannot entirely account for the rate of increase in redundancy observed in Figure 4. Part of the increase is due to the fact that message passing latency in a DVM decreases as redundancy increases (Williams, 2012). Since self-replication time decreases with latency, latency decreases as redundancy increases, and redundancy is increased by self-replication, the process is self-reinforcing. In effect, the actors comprising an SRDVM form an *autocatalytic set*; as the reactant concentration (redundancy) increases, the reaction rate (efficiency) increases correspondingly.

Robust Self-Assembly of Address Sorted Loops

In this section we describe a method for further increasing the efficiency of message passing in DVMs. This is accomplished by representing the heap as an *address sorted loop.*

In order to implement address sorted loops, the actors which comprise DVMs are augmented in several ways. First, they are given a pair of bonds; the *next* bond links an actor to the actor which follows it in the loop, while the *prev* bond links it to the actor which precedes it. Second, every actor is given three additional address registers. The first two (*max* and *min*) hold the maximum and minimum addresses in the loop to which the actor belongs; the third is a *backup* copy of the actor's own address.

The loop is maintained in address sorted order by a low-level behavior implemented by all actors, namely, if an actor's address is greater than the address of the *next* actor (or less than the address of the *prev* actor) then the two actors *swap* positions in the loop. When two actors are swapped, all type and state information is copied from one position to another; bonds are unaffected. The address sorting behavior is modified for the pair of actors with minimum and maximum addresses. These actors do not swap positions but instead serve as anchors for the loop.

A second low-level behavior maintains the maximum and minimum address values. This is accomplished by setting an actor's *max* value equal to the maximum of its neighbors' *max* values (a similar process updates *min*).

Address sorted loops self-assemble by a process which adds one actor at a time in order of increasing address. The self-assembly process is initiated when the actor with address one sees the actor with address two in its neighborhood and forms a length two loop by creating a pair of *prev* and *next* bonds. The *min* and *max* fields of both actors are assigned the values one and two.

When any actor in a loop sees an unbonded actor with address equal to *max* plus one in its neighborhood, it *splices* that actor into the loop; the new actor is moved to a position midway between the actor doing the splicing and the actor which follows (if there is not enough room the action fails). Appropriate surgeries are then performed on the actor's *next* bond (and the *prev* bond of the actor which follows). In this way, the new actor is incorporated into the loop. The low-level sorting behavior possessed by all actors then rapidly moves it to its correct position.

The current *max* address value is also rapidly updated. Due to the non-zero latency required to update the *max* address value, it occasionally happens that two actors with the same address are added to a loop. To deal with this contingency, yet another low-level behavior is used to *eject* one of any two adjacent actors with duplicate *backup* addresses. This can only happen if the gap which would result from the ejection is short enough to be spanned by a bond; if not, the action fails.

Although we experimented with a more complex method of loop self-assembly involving merging of loops with overlapping address ranges followed by ejection of duplicates, the simple self-assembly method of adding one actor at a time turned out to be the most efficient. This is due to the fact that the net diffusion constant of a set of actors linked by bonds rapidly decreases with the size of the set. By comparison, the diffusion constant of an unbonded actor is enormous. In the simple self-assembly method, the length of the loop is analogous to the surface area of a growing sponge. Like a sponge, it is immobile. However, as the loop grows, it becomes more efficient at collecting an actor with the address it needs to extend itself; the self-assembly process actually speeds up. The time (measured in average number of updates per site) required for loop self-assembly decreases as redundancy increases (Figure 5).

The DVM begins execution when the root actor is spliced into the loop. Message passing is accomplished by a simple process in which the sender temporarily changes its address to the address of the recipient. The low-level sorting behavior then moves the sender to a position in the loop adjacent to the recipient, at which point, the message is delivered. Afterwards, the sender restores its own address (using its *backup* copy) and the low-level sorting behavior rapidly returns it to its original position.

Since swapping two actors joined by a bond does not depend on the relative positions of the actors in the grid (unlike splicing and ejection which fail if the bonds which would result are too short or too long), the message passing process is relatively efficient, taking time proportional to the length of the loop. Experiments will show that the speedup relative to diffusion-based message passing is significant.

Recall that evaluation of a Skeme expression requires three different stacks comprised of pairs that are stored in the heap. In fact, these pairs, which are created by bytecode actors (not by the *cons* primitive function) form the bulk of the heap at any given time. Other *transient* objects, *e.g.,*

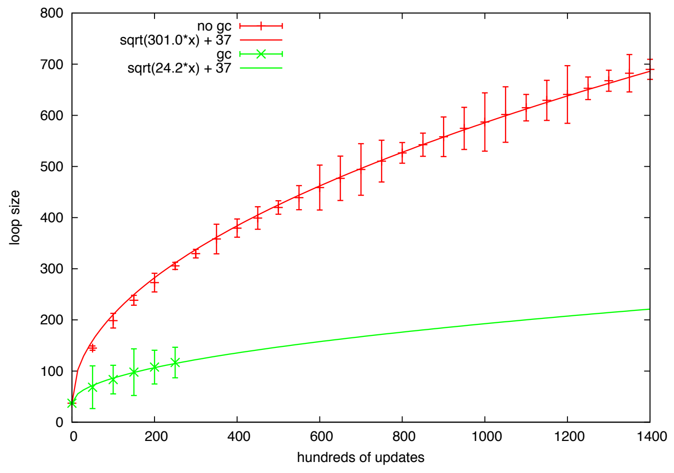

Figure 6: Average loop size versus time for reflection-based SRDVM with (green) and without (red) garbage collection and best fit functions of the form $f(x) = \sqrt{ax} + 37$. The average is over ten runs. Grid size was 128×128. Error bars show $\pm 10\ \sigma$ (for clarity). The SRDVM with garbage collection replicates five times faster.

closures, are also created during evaluation. The fact that message passing latency is proportional to heap size suggests that an aggressive incremental garbage collection process that ejects transient objects from loops when they are no longer needed would yield significant increases in evaluation efficiency.

As part of the implementation of the garbage collection process, the heap's address range is divided into two parts. The first part, consisting of positive addresses, contains permanent objects that form the bytecodes and operands of the mother and daughter object code programs. The second part, consisting of negative addresses, contains transient objects allocated by the DVM during execution. Unbonded actors with negative addresses immediately delete themselves.

When an actor representing a heap-allocated object is created, it is spliced into the loop by bisecting the *next* bond of its creator. Actors created by most primitive functions have positive addresses; actors created by bytecodes (pairs forming the VM's three stacks and closures) have negative addresses. Upon creation, actors with positive addresses are ejected since they duplicate actors already in the DVM; actors with negative addresses are retained. Consequently, during execution, the size of the positive part of the heap stays constant, while the negative part steadily grows. After ejection, actors with positive addresses self-assemble into new address sorted daughter loops.

Recall that the VM's *frame* and *argument* stacks are popped after the application of a primitive function or closure. Whenever a stack is popped, the pairs that comprised the popped record are marked for ejection. Experiments show that this process yields significant increases in eval-

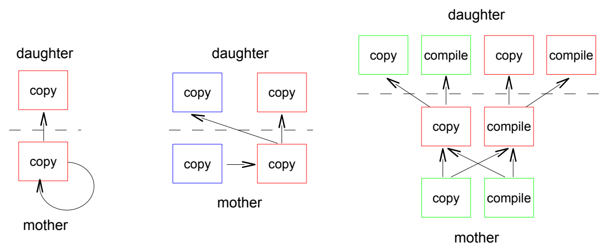

Figure 7: Reflection-based self-replicating program (left) copies itself by exploiting program-data equivalence. Quine self-replicating program (center) with object code genome (blue) and phenome (red). Quine self-replicating program (right) with source code genome (green) and object code phenome (red).

uation efficiency. Indeed, in the case of the reflection-based SRDVM, both the maximum loop size and execution time (measured in average number of updates per site) are five times larger without garbage collection. See Figure 6.

A final behavior only plays a role at the end of execution: when an actor is missing its *next* bond, it deletes its *prev* bond (and vice versa). Since the *halt* bytecode deletes its *next* bond when it receives a continuation, loops rapidly dis-assemble when the DVMs they host finish executing. Un-bonded actors with positive addresses (the bytecodes and operands formerly comprising the mother program) are free to join self-assembling daughter programs; unbounded actors with negative addresses immediately delete themselves.

The life-cycle can be rendered graphically (Williams, 2014). To minimize clutter, only bonds are displayed. The loops' address ranges are mapped to hue so that the effect of the sorting behavior can be verified. The thing that is most striking is the dynamism: everything is moving; nothing is static. SRDVMs look like crumpled rainbows, rapidly assuming different conformations as the actors comprising them undergo diffusion. Continuations move clockwise and counterclockwise inside address-sorted loops that steadily grow, then quickly dissassemble when execution ends.

Self-Replication by Quines

We previously extended Skeme by adding functions that enabled a program to copy heap-allocated objects. In order to define an expression that can recursively copy a binary tree representation of object code, we now extend Skeme with a set of functions that make instances of bytecode types, *e.g.*, the primitive function *make-frame* to make bytecodes of type *frame*. For convenience, we also introduce a primitive function, *make*, which when applied to a bytecode, returns the primitive function that makes instances of that type. Together, these functions make it possible to define an expression that recursively copies object code. One might wonder whether this expression could be compiled and applied to *itself*, yielding a self-replicating object code program. Alas, this is not possible, since it would require that the program

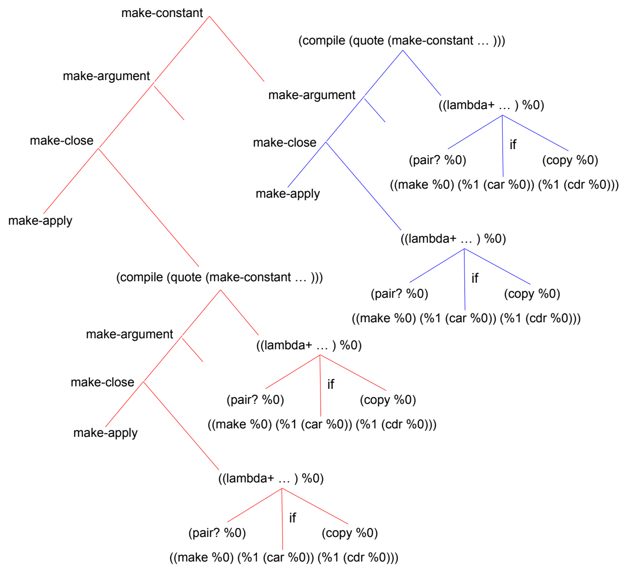

Figure 8: Skeme source code that constructs a self-replicating object code program consisting of 327 bytecodes and operands. Both phenome (red) and genome (blue) are object code programs. Translation and transcription are implemented by identical expressions that copy object code.

contain a *cycle* and cycles cannot be created in pure functional code; Moreover, even if a cycle could be created, there would be no way for the program to know when to stop copying; a more subtle approach is required (Figure 7).

A *quine* is a program that prints itself. All quines consist of two parts. Conventionally called *program* and *data*, they may be thought of as *phenome* and *genome*. All quines work the same way. Active program transforms passive data in two ways producing new instances of both program and data. Equivalently, the *mother* quine's genome is *transcribed* and *translated* yielding the *daughter* quine's genome and phenome. The forms of the genome and phenome, and the nature of the translation and transcription processes, differ from quine to quine. [Hasegawa and McMullin (2013) recently defined a quine inside Avida. To our knowledge, this is the first time this has been done.]

Quines can be written in any programming language but Skeme's list-based syntax, together with quotation, make it easy to write an especially short and simple one. In the following Skeme quine, phenome is an expression that evaluates to a closure that appends a value to the same value quoted; genome is just phenome quoted. Finally, phenome is applied to genome:

```
((lambda (list %0 (list quote %0)))
 (quote (lambda (list %0 (list quote %0)))))
```

It is possible to write an object code quine that works much the same way. In place of the special-forms *lambda* and *quote*, the object code quine uses the bytecodes *close* and *constant*. Instead of building a function application

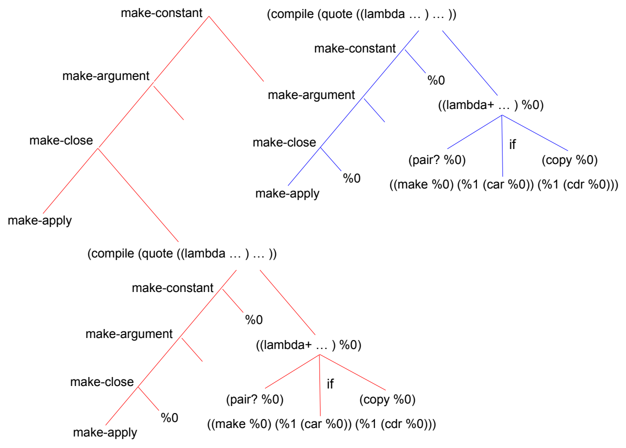

Figure 9: The fact that the translation and transcription processes both return the copied genome can be exploited by introducing a name for this value with a *lambda* expression. The result is a more efficient self-replicating program.

in source code with *list*, the equivalent is built in object code using *make-argument* and *make-apply*. Finally, while the source code quine doesn't actually copy its components (since references suffice for printing) the object code quine must (since the goal is self-replication). The self-similarity of the object code quine can be appreciated by inspecting the Skeme source code that is used to build it (Figure 8). The four bytecodes that comprise the quine's backbone create a closure and apply it to a quoted copy of its own body. These actions build object code that will itself (when executed) create and apply a closure to a quoted copy of its own body; that is, will construct another copy of the quine.

The quine thus constructed contains 327 bytecodes and operands. However, it is needlessly inefficient since it copies its object code genome twice using identical translation and transcription processes. It can be made significantly smaller and more efficient by introducing a name for the value of the copied genome with a *lambda* expression (Figure 9). This quine copies its genome once but uses the copy twice (once as genome and once as phenome) and contains only 107 bytecodes and operands. Its replication rate is contrasted with that of the reflection-based SRDVM in Figure 10.

A self-hosting compiler compiles the same language it is written in. Consequently, it can compile *itself*. It is possible to define a very short self-hosting compiler φ for Skeme (Figure 11). Inserting a copy of φ into the unquoted half of the Skeme quine (phenome) so that it compiles its result and mirroring this change in the quoted half (genome) yields

```
((lambda (φ (list %0 (list quote %0))))
 (quote (lambda (φ (list %0 (list quote %0))))))
```

which, although not a quine itself, returns a quine when evaluated; this quine is not a source code fixed-point of the Skeme interpreter but an object code fixed-point of the Skeme VM. In effect, it is a quine in a low-level language

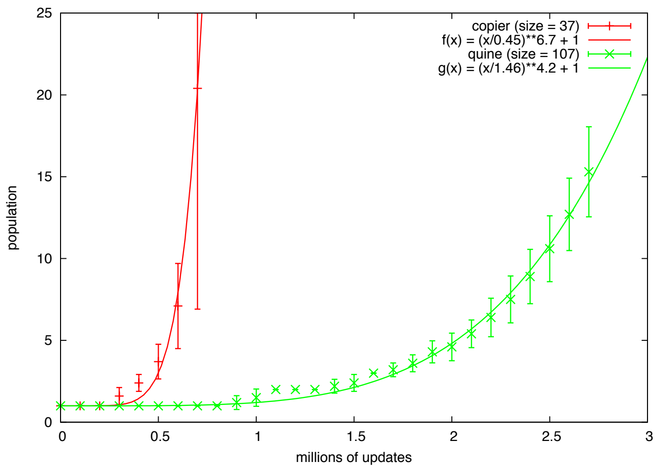

Figure 10: Average population versus time for 37 bytecode reflection-based SRDVM (red) and 107 bytecode quine-based SRDVM (green) and best fit functions of the form $(x/a)^b + 1$. The averages are over ten runs. Grid size was 256×256. Error bars show $\pm\sigma$.

(phenome) that reproduces by compiling (translation) and copying (transcription) a compressed self-description written in a high-level language (genome). When reified, the SRDVM consists of 990 actors representing a mixture of source and object code. It has been verified that it replicates perfectly across generations and simple statistics including maximum heap and loop sizes, and execution time (measured in average number of updates per site) have been determined for it and for the other SRDVMs. See Table 2.

Table 2: A comparison of four SRDVMs.

code size	max heap size	max loop size	execution time
37	7.2×10^2	1.3×10^2	2.7×10^4
107	1.7×10^3	3.7×10^2	2.7×10^5
327	5.5×10^3	1.1×10^3	2.2×10^6
990	1.8×10^4	5.0×10^3	2.5×10^7

Future Work

Before SRDVMs can be used in the artificial life approach to evolutionary computation several unsolved problems must be addressed. These include the isolation of genomes, reproduction with variation, and self-replication efficiency.

Membranes were essential to the evolution of complex life. Without membranes to concentrate reactants and isolate genomes, neither metabolism nor evolution would be possible. The address sorted loops described here concentrate reactants quite effectively, but do not (currently) provide a mechanism for the isolation of genomes.

If SRDVMs are to evolve, then they must not only isolate their genomes, they must reproduce with variation. Artificial organisms in Tierra and Avida do this because the host sys-

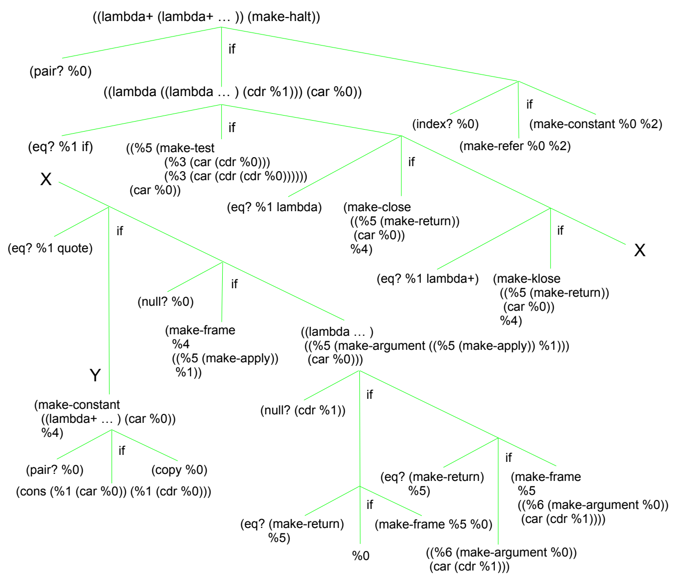
((lambda+ (lambda+ …)) (make-halt))
if
(pair? %0)
((lambda ((lambda …) (cdr %1))) (car %0))
(index? %0)
if
(make-constant %0 %2)
(make-refer %0 %2)
if
(eq? %1 if)
((%5 (make-test (%3 (car (cdr %0))) (%3 (car (cdr (cdr %0)))))) (car %0))
X
if
(eq? %1 lambda)
(make-close ((%5 (make-return)) (car %0)) %4)
if
X
(eq? %1 quote)
if
(eq? %1 lambda+)
(make-klose ((%5 (make-return)) (car %0)) %4)
(null? %0)
if
(make-frame %4 ((%5 (make-apply)) %1))
((lambda …) ((%5 (make-argument ((%5 (make-apply)) %1))) (car %0)))
Y
(make-constant ((lambda+ …) (car %0)) %4)
if
(null? (cdr %1))
if
(pair? %0)
(copy %0)
(cons (%1 (car %0)) (%1 (cdr %0)))
(eq? (make-return) %5)
if
(make-frame %5 ((%6 (make-argument %0)) (car (cdr %1))))
(eq? (make-return) %5)
if
(make-frame %5 %0)
%0
((%6 (make-argument %0)) (car (cdr %1)))

Figure 11: Self-hosting compiler for Skeme can be used to construct an SRDVM comprised of 990 actors of mixed source and object code. The compiler defines the *genotype-phenotype mapping* of a new kind of artificial organism with source code genome and object code phenome.

tems introduce random changes in the instructions of hosted organisms. Spector and Robinson (2002) describe a compelling alternative approach where artificial organisms not only manage their own replication, but also the mutation of their genomes. For SRDVMs to reproduce with variation in this way, a set of mutation operators needs to be defined in Skeme and these operators need to be employed in the subtree of the compiler that copies the genome.

The last problem that must be solved is efficiency. There is nearly a thousandfold difference between the shortest and longest execution times in Table 2. Given that a DVM requires O(M^2) time to build a heap of size M, this difference is understandable. Nevertheless, before they can be useful, SRDVMs must be efficient, and although O(1) message passing latency is not possible in a machine that isn't random access, we believe that O($\sqrt{M}$) is, and this would be a significant improvement.

Conclusion

The breadth of research in artificial life ranges from genetic programming in Lisp to the engineering of biochemical protocells. It might seem that the gulf between these topics is so large that it will never be spanned. Yet many stepping stones already exist. The concept of *fixed-points in a chemical lambda calculus* described by Fontana and Buss (1994) bridged part of the gulf in the author's own mind several years ago and inspired this paper. By introducing DVMs to the artificial life community and demonstrating an SRDVM that replicates by compiling its own source code, this paper attempts to place another stepping stone in the gulf. It does so by suggesting that the artificial life approach to evolutionary computation exemplified by systems like Tierra and Avida might be pursued using self-replicating programs written in high-level languages hosted on a computational substrate that has the dual virtues of making competition between programs for all uses of memory zero sum and being indefinitely scalable.

Acknowledgements

The author wishes to thank Dave Ackley, Tom Hayes, and Barry McMullin for helpful conversations.

References

Ackley, D. (2013). Bespoke physics for living technology. *Artificial Life*, 19(3-4):347–364.

Adami, C., Brown, C. T., and Kellogg, W. (1994). Evolutionary learning in the 2D artificial life system "Avida". In *Artificial Life IV*, pages 377–381. MIT Press.

Berman, P. and Simon, J. (1988). Investigations of fault-tolerant networks of computers. In *STOC*, pages 66–77.

De Bruijn, N. G. (1972). Lambda calculus notation with nameless dummies: a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34:381–392.

Dybvig, R. K. (1987). *Three implementation models for Scheme*. PhD thesis, University of North Carolina.

Fontana, W. and Buss, L. W. (1994). What would be conserved if the tape were played twice? *Proc. Natl. Acad. Sci. USA*, 91:757–761.

Hasegawa, T. and McMullin, B. (2013). Exploring the point-mutation space of a von Neumann self-reproducer within the Avida world. In *ECAL*.

Hewitt, C., Bishop, P., and Steiger, R. (1973). A universal modular actor formalism for artificial intelligence. In *IJCAI*.

Landin, P. J. (1964). The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320.

Nakamura, K. (1974). Asynchronous cellular automata and their computational ability. *System Comput. Controls*, 15(5):56–66.

Nehaniv, C. L. (2004). Asynchronous automata networks can emulate any synchronous automata network. *IJAC*, 14(5-6):719–739.

Ray, T. S. (1994). An evolutionary approach to synthetic biology, Zen and the art of creating life. *Artificial Life*, 1:179–209.

Spector, L. and Robinson, A. (2002). Genetic programming and auto-constructive evolution with the Push programming language. *Genetic Programming and Evolvable Machines*, 3(1):7–40.

Williams, L. R. (2012). Robust evaluation of expressions by distributed virtual machines. In *UCNC*.

Williams, L. R. (2014). Self-replicating distributed virtual machines (video supplement). In `http://www.cs.unm.edu/~williams/srdvm.mov`.