Evolution of Tail-Call Optimization in a Population of Self-Hosting Compilers

Lance R. Williams[1]

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

Abstract

We demonstrate the evolution of a more complex and more efficient self-replicating computer program from a less complex and less efficient ancestor. Both programs, which employ a novel method of self-replication based on compiling their own source code, are significantly more complex than programs which reproduce by copying themselves, and which have only exhibited evolution of degenerate methods of self-replication.

Introduction

Among living organisms, which employ many and varied mechanisms in the process of reproduction, examples of evolved mechanisms which are both more complex and more efficient than ancestral mechanisms, abound. Yet, nearly twenty years after (Ray, 1994)'s groundbreaking work on the Tierra system, in which the evolution of many novel (but degenerate) methods of self-replication was first demonstrated, there is still no example of a more complex and more efficient self-replicating computer program evolving from a less complex and less efficient ancestor.

This is not to say that there has been no progress in the field of artificial life since Tierra. Nor are we suggesting that increased reproductive efficiency is the only evolutionary path to increased complexity. The evolution of self-replicating programs of increased complexity has been demonstrated many times(Koza, 1994; Taylor and Hallam, 1997; Spector and Robinson, 2002), and perhaps most convincingly in the Avida system(Adami et al., 1994). However, more complex programs evolved in Avida only because complexity was artificially equated with efficiency in the sense that programs which learned to solve problems unrelated to self-replication were rewarded with larger rations of CPU time. No program in Avida (or in any other system known to us) has ever evolved a method of self-replication that is both more complex and more efficient than the method employed by its ancestor.

A New Kind of Artificial Organism

Self-replicating programs have been written in both high-level languages and machine languages. We define a machine language program to be *interesting* if it prints a string at least as long as itself and halts when executed, and observe that the Kolmogorov complexity of interesting programs is lower than that of random strings of similar length. Now, if we were to train an adaptive compression algorithm on a large set of interesting programs, then the compressed programs which result would *look* like random strings. However, by virtue of being shorter, they would be more numerous relative to truly random strings of similar length. It follows that compression, which *decreases* redundancy by replacing recurring sequences of instructions with invented names, *increases* the density of interesting programs.

Since both processes increase redundancy and output machine language programs, it is natural to identify *decompression* with *compilation*, which increases redundancy by repeatedly generating similar sequences of instructions while traversing a parse tree. Viewed this way, programs written in (more expressive) high-level languages are compressed machine language programs, and compiling is the process of decompressing source code strings into object code strings which can be executed by a CPU.

If the density of interesting programs increases with the expressiveness of the language in which they are encoded (as the above strongly suggests), then one should use the most expressive language possible for any process, like genetic programming, which involves searching the space of interesting programs. However, if the goal is building artificial organisms, then high-level languages have a very serious drawback when compared to machine language. Namely, programs in high-level languages must be compiled into machine language before they can be executed by a CPU or be reified as a *distributed virtual machine*(Williams, 2012).

Given that we want our self-replicating programs to be both (potentially) reifiable and to evolve into programs of greater complexity and efficiency, we must ask: How can the advantages which derive from the use of a high-level language for genetic programming be reconciled with the fact that only machine language programs can be reified?

To address this question, we introduce a new and significantly more complex kind of artificial organism–a ma-

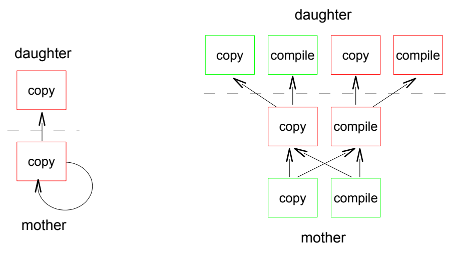

Figure 1: Conventional self-replicating program (left) copies itself by exploiting program-data equivalence of von Neumann architecture. Compiling quine self-replicating program (right) with source code genotype (green) and object code phenotype (red). Because the shortest correct implementation of copy is optimal, only the compiling quine is capable of non-degenerate evolution.

chine language program which reproduces by compiling its own source-code. See Figure 1. Conventional self-replicating programs reproduce by copying themselves. Optimum copiers accomplish this in time proportional to their length, and it is not very hard to write a copier which is optimum in this sense (or for one to evolve). It follows that shorter implementations are always more efficient, which leads to degenerate evolution, absent factors beyond efficiency. The possible variation in the implementation of a compiler is far larger. Even if the definition of the object language is stipulated, there is still a huge space of alternative implementations, including the syntax and semantics of the source language, the ordering of the decision tree performing syntactic analysis, and the presence (or absence) and effectiveness of any object code optimizing procedures.

In this paper we describe a machine language program which reproduces by compiling its own source code and use genetic programming to demonstrate its capacity for non-degenerate evolution. In the process we address questions such as: How can a program like a compiler, which implements a complex prescribed transformation, evolve improvements while avoiding non-functional intermediate forms? How can two lexically scoped programs be combined by crossover without breaking the product? How can a more efficient self-replicating program evolve from a less efficient ancestor when all mutations initially yield higher self-replication cost?

A Simple Programming Language

Because a self-hosting compiler compiles the same language it is written in, it can compile itself. The language we used to construct our self-hosting compiler 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 run-time 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).

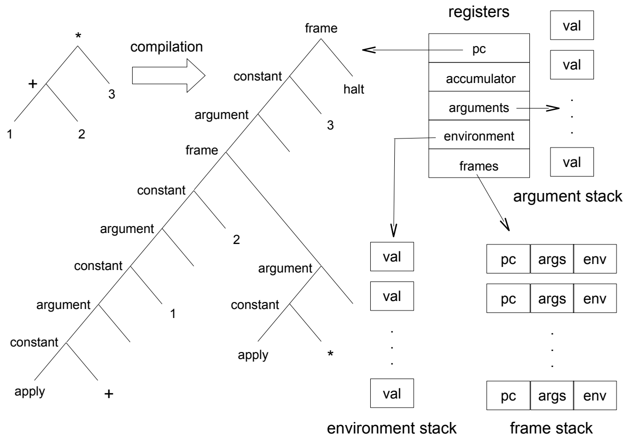

Figure 2: Virtual machine for evaluating compiled Scheme expressions showing its registers and associated heap-allocated data structures(Dybvig, 1987).

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.

Tail-Call Optimization

Because the very first self-hosting compiler was written in Lisp, it is not surprising that it is possible (by including primitive functions which construct bytecode types) to write a very small self-hosting compiler in Skeme. See Figures 2 and 3.

The cost of compiling a given source code depends not only on its size, but also on the complexity of the source language, the efficiency of the compiler, and the cost of any object code optimizations it performs. Common compiler optimizations include constant folding, loop unrolling, function inlining, loop-invariant code motion, elimination of common subexpressions, and dead code elimination. Since a self-hosting compiler compiles *itself*, the efficiency of the object code it *generates* also affects compilation cost; it follows that minimizing the cost of self-compilation involves a complex set of tradeoffs. The most important of these is that object code optimizations must pay for themselves by yielding an increase in object code efficiency large enough to offset the additional cost of compiling the source code implementing the optimization.

Most of the overhead associated with a function call involves the saving and restoration of evaluation contexts. In Skeme, these operations are performed by the *frame* and *return* bytecodes which push and pop the frame stack. However, when one function calls another function in a *tail position*, there is no need to save an evaluation context, because the restored context will just be discarded when the first function returns. A compiler which performs *tail-call optimization* recognizes when a function is called in a tail position and does not generate the code which saves and restores evaluation contexts. This not only saves time, it also saves space, since tail recursive function calls will not increase the size of the frame stack at runtime.

A Quine which Compiles Itself

A *quine* is a program which prints itself. It is possible to write a quine in any programming language but Skeme's list-based syntax makes it possible to write especially short and simple quines. For example, in the following Skeme quine, an expression (lambda (list %0 (list quote %0))) which evaluates to a closure which appends a value to the same value quoted is applied to the same expression quoted:

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

It is possible to define an expression φ in Skeme which can compile any Skeme expression. The expression φ evaluates to a curried function which takes a compiled expression and an uncompiled expression as arguments. The compiled expression is a *continuation*; the uncompiled expression is the source code to be compiled; applying the curried function to the halt bytecode yields a function which can compile top-level expressions. Inserting a copy of (φ (make-halt)) into the unquoted half of the quine so that it compiles its result (and mirroring this change in the quoted half) yields

```
((lambda ((φ (make-halt))
          (list %0 (list quote %0))))
 (quote (lambda ((φ (make-halt))
                 (list %0 (list quote %0))))))
```

which, although not a quine itself, returns a quine when evaluated. Significantly, this quine is not a source code fixed-point of the Skeme interpreter but an object code fixed-point of Dybvig's virtual machine. In effect, it is a quine in a low-level language (phenotype) which reproduces by compiling a compressed self-description written in a high-level language (genotype).

In prior work on evolution of self-replicating programs there has been no distinction between phenotype and genotype; mutations are made on the same representation which is evaluated for fitness. In contrast, in living organisms, small changes in genotype due to mutation can be amplified by a development process and result in large changes in phenotype; it is phenotype which is then evaluated for fitness. In a compiling quine, small changes in source code (genotype) are amplified by compilation (development) yielding much larger changes in object code (phenotype) and it is object code which determines fitness, since its execution consumes the physical resources of space and time.

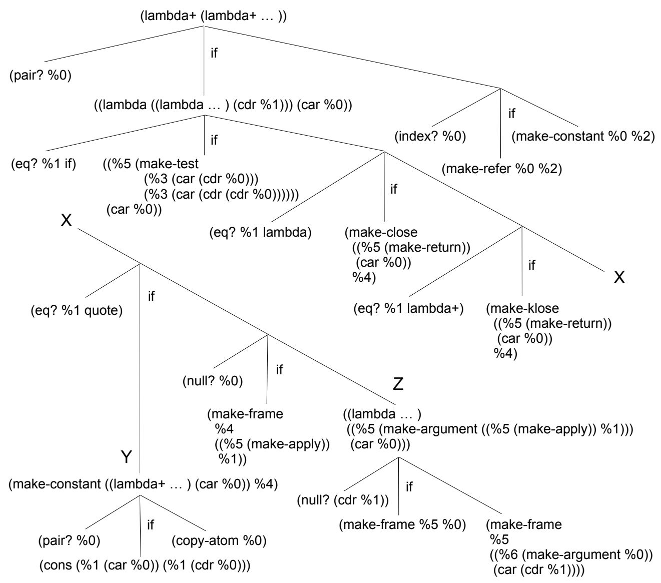
(lambda+ (lambda+ …))
if
(pair? %0)
((lambda ((lambda …) (cdr %1))) (car %0))
if
(index? %0)
(make-constant %0 %2)
(make-refer %0 %2)
if
(eq? %1 if)
((%5 (make-test (%3 (car (cdr %0))) (%3 (car (cdr (cdr %0)))))) (car %0))
X
if
(eq? %1 lambda)
(make-close ((%5 (make-return)) (car %0)) %4)
if
(eq? %1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

Figure 3: An expression φ for compiling Skeme into object code able to compile itself. The X indicates a break in the figure; the subtree labeled Y copies the Skeme source code and the subtree labeled Z compiles function applications.

Related Work

(Stephenson et al., 2003) described a genetic programming system which learns priority functions for compiler optimizations including hyperblock selection, register allocation, and data prefetching. (D'haeseleer, 1994) described and experimentally evaluated a method for context preserving crossover. (Kirshenbaum, 2000) demonstrated a genetic programming system where crossover is defined so that it respects the meaning of statically defined local variables.

Several authors have explored the idea of staged or alternating fitness functions. (Koza et al., 1999) used a staged fitness function as a method for multi-objective optimization. (Pujol, 1999) described a system where the fitness function is switched after a correct solution is discovered to a function which minimizes solution size. (Zou and Lung, 2004) and (Offman et al., 2008) used alternating fitness functions to preserve diversity in genetic algorithm derived solutions to problems in water quality model calibration and protein model selection.

Genetic Programming

Our approach to genetic programming is motivated by the fact that gene duplication followed by specialization of one or both copies is a common route to increased complexity in biological evolution(Finnigan et al., 2012). We introduce

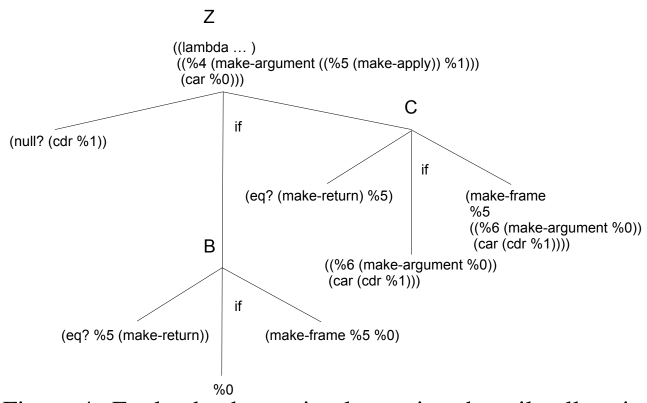

Figure 4: Evolved subtrees implementing the tail-call optimizations which characterize the B and C genotypes. The A genotype performs neither optimization while the D genotype performs both. Both optimizations check to see if the continuation is a return bytecode, which performs a frame stack pop. If so, the push-pop sequence is not generated, resulting in significant savings in time and space usage.

two mutation operators called *bloat* and *shrink* which play roles analogous to gene duplication and specialization and employ these in a genetic programming system where fitness alternates between object code based definitions of complexity and self-replication efficiency. In teleological terms, the bloat operator attempts to increase complexity by adding source code while the shrink operator attempts to increase self-replication efficiency by removing it.

Alternating Fitness Function

Time is divided into ten generation periods termed *epochs* which alternate between two types, *flush* and *lean*. In flush epochs, fitness is defined as *effective complexity* while in lean epochs it is defined as *self-replication efficiency*.

A test bytecode is defined to be *non-trivial* if both of its continuations are exercised in the course of self-replication. This will only happen if the predicate expression in the *if* special-form from which the test bytecode is compiled sometimes evaluates to true and sometimes to false. The number of non-trivial test bytecodes in the object code is a good measure of the source code's effective complexity. Consequently, in flush epochs the number of non-trivial test bytecodes in the object code is *maximized*.

Because frame stack pushes and pops are the most expensive operation performed by the virtual machine, they are an excellent proxy for overall self-replication cost. Consequently, in lean epochs, the number of frame stack pops, which are implemented by the return bytecode, is *minimized*.

Mutations can be classified as beneficial, neutral, harmful, and lethal. The purpose of the bloat operator is to introduce source code which can be shaped by the shrink operator and by crossover. Significantly, the introduced code does not change the value of any expression which contains

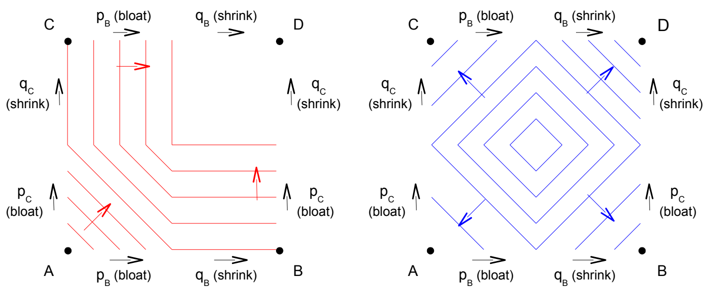

Figure 5: Contour plots of fitness landscapes during flush (left) and lean (right) epochs. Colored arrows point in directions of increased fitness. In lean epochs, the four genotypes A, B, C, and D occupy islands separated by valleys of decreased fitness; the bloat mutations necessary for A to evolve into any of the other genotypes are harmful since they increase the cost of self-replication. In contrast, the shrink mutations required for A to evolve into any of the other genotypes are beneficial. In flush epochs, the situation is reversed–the bloat mutations are beneficial and the shrink mutations are harmful since they increase and decrease effective complexity respectively. Alternating between the two fitness functions creates paths between the A and D genotypes consisting solely of beneficial mutations.

it; it is *value-neutral* with respect to evaluation. Because (by their nature) they increase the cost of self-replication without breaking the compiler, bloat mutations (although never lethal) are harmful during lean epochs.

In contrast, shrink mutations are beneficial when they reverse bloat mutations during lean epochs and can be harmful when they reverse bloat mutations during flush epochs. However, shrink mutations have two different and more pronounced effects. First, a shrink mutation can remove code and break the compiler, in which case it is lethal. Second, it can shape the result of a bloat mutation in a way which decreases the cost of self-replication, in which case it will be strongly beneficial during lean epochs and become fixed in the population.

Bloat

The source code for the self-hosting compiler contains boolean-valued expressions with six different syntactic forms. Excluding primitive functions, the source code contains six different expressions of constant value. A random syntactic form can be combined with a random de Bruijin index and (if necessary), a random constant-valued expression, to construct a random boolean-valued expression, ϕ.

The bloat operator is defined by five rules. The first four rules define a recursive procedure which applies the bloat operator in selected contexts. The last rule replaces a function application with an if expression which returns the same value regardless of whether a random boolean-valued expression, ϕ, evaluates to true or false. Consequently, the value of the expression is the same before and after the mutation. The fact that the bloat operator is value-neutral with respect to evaluation is important because only viable individuals (those which correctly self-replicate) are copied to the next generation; and although a bloat mutation typically introduces expressions which are not evaluated during self-replication (which greatly reduces the fitness of affected individuals by increasing their self-replication costs) affected individuals always remain viable because bloat mutations cannot actually break the compiler which contains them. The five rules which define the bloat operator are

1. (lambda[+] e_1) $\rightarrow$ (lambda[+] e'_1)
2. ((lambda[+] e_1) e_2) $\rightarrow$ ((lambda[+] e'_1) e'_2)
3. (if e_1 (id e_2) e_3) $\rightarrow$ (if e_1 (id e_2) e_3)
4. (if e_1 e_2 e_3) $\rightarrow$ (if e_1 e'_2 e'_3)
5. (f $e_1 \ldots e_N$) $\rightarrow$ (f $e_1 \ldots e_N$) || (if ϕ (id (f $e_1 \ldots e_N$)) (f $e_1 \ldots e_N$))

where f is a primitive function, ϕ is a random boolean-valued expression, *id* is the identity function, and primes mark expressions which are recursively expanded. Alternative right hand sides are separated by vertical bars; the alternative to the left of the || (no mutation) is chosen with 95% probability; the remaining alternative (mutation) is chosen otherwise. The identity function serves as a value neutral tag in a meta-syntax; because the third rule has the same left and right hand sides, the recursive procedure which applies the bloat operator will not descend into if subtrees marked with this tag; this prevents the compounding of bloat mutations.

Shrink

The rules defining the shrink operator serve two purposes. the first purpose is to reverse mutations introduced by the bloat operator; the fourth shrink rule removes the tagged if expressions generated by the bloat operator so that a bloat mutation followed by a shrink mutation (of this type) has no net effect. The second purpose is to simplify function applications; the last shrink rule replaces an expression where a function is applied to one or more values with just one of those values. Because these rules also remove the identity function tags inserted by the bloat operator, the expression which results from a shrink mutation is again subject to bloating. The five rules which define the shrink operator are

1. (lambda[+] e_1) $\rightarrow$ (lambda[+] e'_1)
2. ((lambda[+] e_1) e_2) $\rightarrow$ ((lambda[+] e'_1) e'_2)
3. (if e_1 e_2 e_3) $\rightarrow$ (if e_1 e'_2 e'_3)
4. (if e_1 (id e_2) e_3) $\rightarrow$ (if e_1 (id e'_2) e'_3) || e_2 | e_3
5. ($fe_1 \ldots e_N$) $\rightarrow$ ($fe_1 \ldots e_N$) || e_1 | $\ldots$ | e_N

Table 1: Complexities and self-replication costs.

	A	B	C	D
non-trivial tests	8	9	9	10
returns	551	333	432	183

where f is a primitive function, *id* is the identity function, and primes mark expressions which are recursively expanded. Alternative right hand sides are separated by vertical bars; the alternative to the left of the || (no mutation) is chosen with 95% probability; one of the remaining alternatives (mutation) is chosen otherwise (each with equal probability). Unlike the bloat operator, which is value neutral, the shrink operator changes the object code generated by the compiler when it modifies an expression which is evaluated during self-replication. In the case of the fourth shrink rule, this often reverses a harmful bloat mutation, in which case the shrink mutation is beneficial. However, in the case of the last shrink rule, the mutation most often breaks the compiler. Very rarely, the shrink mutation does not break the compiler but instead results in a decrease in self-replication cost.

The problem which plagues many genetic programming systems, in which code trees grow larger with increasing time, does not occur for two reasons. First, the use of the *id* function as a tag prevents the bloat operator from being applied within if expressions which were themselves just created. Second, the shrink operator reverses bloat mutations, and bloat mutations not yielding a decrease in self-replication cost are strongly selected against during lean epochs.

The combined effect on fitness of these two mutation operators is complex. After a pair of bloat and shrink mutations, a more complex source code must be analyzed by a more complex compiler, a change which might (but more likely will not) pay for itself by an increase in the efficiency of the generated object code.

Crossover

Because the self-hosting compiler is a complex lexically scoped program, variables which are defined in one scope will not necessarily be defined in other scopes. If we employed the standard method of non-homologous crossover used in most work on genetic programming, then subtrees could be inserted into scopes where one or more variables are undefined, and this would break the compiler. We address this problem by employing the *homologous* crossover method described by (D'haeseleer, 1994). In this method, the crossover operator descends into both parent trees in parallel; points where the two parent trees differ are subject to crossover, with the child receiving the subtree of either parent with equal probability. D'haeseleer notes that homologous crossover facilitates convergence (fixation) since children resulting from the crossover of identical parents will also be identical to the parents.

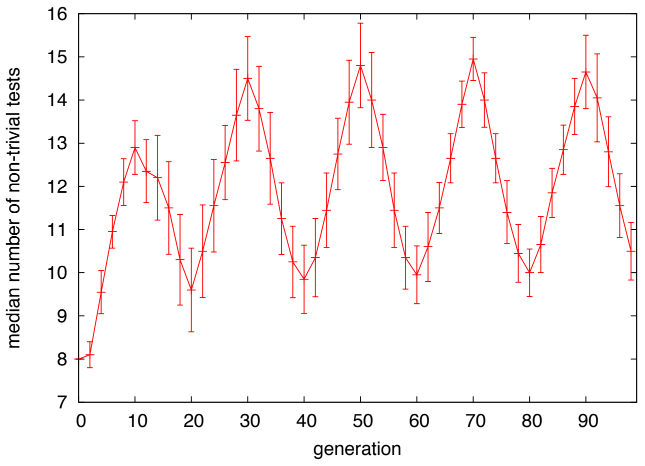

Figure 6: The median number (in a population of size 200) of non-trivial test bytecodes averaged over 20 runs (error bars show plus or minus one standard deviation). Because each non-trivial test bytecode results from a bloat mutation at a distinct point in the φ expression, this graph demonstrates that mutation is in no way restricted to the two points relevant to the evolution of tail-call optimization.

Genotypes

Function applications involving one and two arguments are compiled at two different points in the φ expression and each of these points is a potential target for a pair of bloat and shrink mutations which would partially implement tail-call optimization. We call the genotype of programs which perform neither optimization A, one (or the other) optimization B (or C), and both optimizations, D. Both optimizations check to see if the continuation is a return bytecode, which performs a frame stack pop. If so, the push-pop sequence is not generated, resulting in significant time and space savings. See Figure 4. Lower bounds for the complexity and self-replication cost of each of the four genotypes are shown in Table 1. Finally, the relative fitnesses of the four genotypes are shown graphically, in the context of the fitness landscapes for the flush and lean epochs, in Figure 5.

Experimental Results

The initial population consisted of two hundred identical individuals of genotype A at the beginning of a flush epoch (in which fitness is equated with effective complexity). In the first step of the genetic algorithm, the bloat and shrink operators are applied to all individuals in the population and the mutants which result are tested for viability. To test for viability, the mutant is evaluated to produce a daughter, and the daughter is evaluated to produce a granddaughter. The mutant is classified as viable if the daughter and granddaughter contain the same number (greater than zero) of bytecodes (this is done in lieu of a much more expensive test of actual structural equivalence). Viable mutants replace their pro-

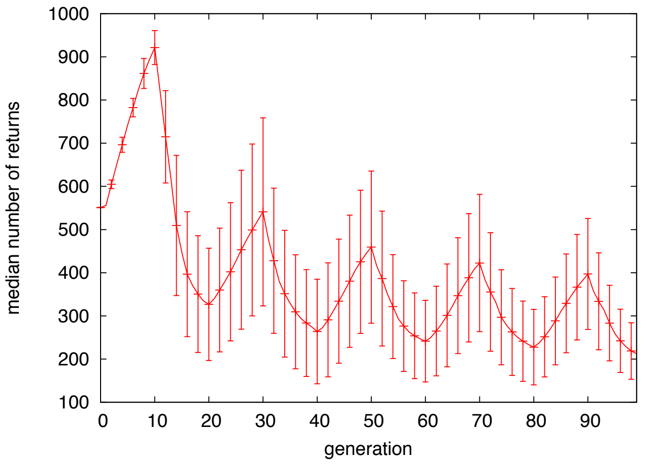

Figure 7: The median number (in a population of size 200) of return bytecodes executed during self-replication averaged over 20 runs (error bars show plus or minus one standard deviation).

genitors in the population.

The population is then subjected to crossover using tournament selection. In each tournament, four individuals are chosen at random (with replacement). The winners of two tournaments are then combined using crossover, and the resulting individual is tested for viability. The crossover operation is repeated until it yields two hundred viable individuals which comprise the population of the next generation.

The above process is repeated for nine more generations, then the epoch is switched to lean (in which fitness is equated with self-replication efficiency). The genetic algorithm is run for a total of 100 generations (five flush epochs interrupted by five lean epochs).

In an initial experiment, the system was run twenty times. The median number of interesting test bytecodes contained in the compiled φ expression and the median number of return bytecodes executed during self-replication were then plotted as a function of generation; see Figures 6 and 7. As expected, both complexity and self-replication cost increase in flush epochs and decrease in lean epochs. After 40 generations (two flush-lean cycles), the median complexity at the end of flush epochs is nearly double its initial value, which means that the majority of individuals contain 7 or more predicates which compile to non-trivial test bytecodes not present in the initial population. Furthermore, the median complexity at the end of lean epochs is always 10 or more, which suggests that either 1) the shrink operator is not fully able to reverse the effects of the bloat operator so that one or more bloat mutations (on average) survive through lean epochs; or 2) one (or both) of the B and C alleles is fixed in the population. Examination of Figure 7 shows that after 40 generations, the median self-replication cost at the end of lean epochs is slightly more than half of its initial value.

This is consistent with evolution of one or both of the B and C genotypes. Self-replication cost continues to increase and decrease (depending on epoch) eventually reaching a point where the median value at the end of the fifth lean epoch is nearly three times smaller than the initial value. This is consistent with the evolution of the D genotype.

After running the system 100 times, the probabilities of the B, C, and D genotypes evolving and for the mutations becoming fixed in the population were estimated. See Table 2. Notably, the most complex and most efficient genotype, D, evolved within 100 generations 81 times. Additionally, the average and median number of generations required for each genotype to evolve and for the mutations to become fixed were also estimated. Considering only the 81 runs in which the D genotype evolved, the average number of generations required was approximately 36 and the median number was 29.

Table 2: Generation of initial evolution and fixation.

	B	C	D	B'	C'	D'
probability	0.90	0.91	0.81	0.89	0.78	0.70
mean	21.8	24.5	35.8	29.9	34.3	43.3
std. dev.	21.0	22.0	24.5	21.1	22.2	24.5
median	11	13	29	17	33	36

If we know the average numbers of individuals of a given genotype in each generation, then we can compute cumulative distribution functions for evolution and fixation of that genotype; see Figure 8. If we examine the c.d.f.'s we see several interesting things.

First, the c.d.f.'s for evolution of genotypes have zero slope during lean epochs, which suggests that new genotypes typically appear during flush epochs, when fitness is equated with effective complexity. Conversely, the c.d.f.'s for genotype fixation have zero slope during flush epochs, which leads us to conclude that fixation of genotypes typically occurs during lean epochs, when fitness is equated with efficiency. This is consistent with an increase in diversity during flush epochs and a decrease during lean epochs.

Second, there is always a lag between the generations of evolution and fixation, and the size of the lag depends on the improvement in self-replication efficiency–the greater the improvement, the shorter the lag. The C allele (which confers an advantage of 119 returns relative to the A allele) requires more time for fixation than the B allele (which confers an advantage of 218 returns).

If we know the generation in which each genotype evolved, it is possible to estimate probabilities for each of the pathways leading from the (least complex and least efficient) A genotype to the (most complex and most efficient) D genotype; see Table 3. This analysis shows that in 64% of the runs in which D evolved, one of the B or C alleles evolved and was fixed prior to the evolution of the other; the D genotype then evolved by mutation from an ancestral program of the B or C genotype. However, in 35% of the runs in which D evolved, something (arguably) more interesting happened. Namely, the B and C alleles evolved in distinct lineages before either was fixed. The D genotype then evolved when an individual with the B allele and an individual with the C allele were combined by crossover. Stated differently, in 35% of the runs where D evolved, beneficial traits which evolved separately were combined by crossover to produce a child program more complex and more efficient than either parent program.

Table 3: Probabilities of pathways to D genotype.

$t_B < t_C = t_D$	$t_C < t_B = t_D$	$t_B < t_C < t_D$	$t_C < t_B < t_D$
0.33	0.31	0.26	0.09

Future Work

This paper describes work that, although preliminary, opens many avenues for further exploration, including

- Determining whether or not a self-replicating program which reproduces by compiling itself can evolve the optimum order for the tests comprising the decision tree which performs syntactic analysis; this would require a new mutation operator which can reorder nested-*if* expressions.
- Determining whether or not it is possible to evolve dead code elimination, which would be a useful optimization in a system which includes mutation operators (like bloat) which (in effect) introduce dead code; to accomplish this, the bloat operator would have to generate a much larger set of ϕ expressions, including dereferencing source code with *car* and *cdr* combinations.
- In the present system, de Bruijn indices are used mainly to simplify the compilation process by eliminating the need for static analysis; however, it is difficult to see how new lexical scopes could evolve (via a new mutation operator which introduces *lambda* expressions) unless bound variables are represented by symbols, and this would mean that the self-hosting compiler must be generalized so that it performs static analysis.
- Demonstration of *auto-constructive evolution* as described by (Spector and Robinson, 2002), in which artificial organisms possess not only their own means of self-replication, but also of producing variation; this would require coding all mutation operators in Skeme and including this code in the subtree of the self-hosting compiler which copies quoted expressions.
- Reification of the compiling quine as a self-replicating distributed virtual machine (including the items listed above) and demonstration of evolution of increased complexity and self-replication efficiency by reified artificial organisms.

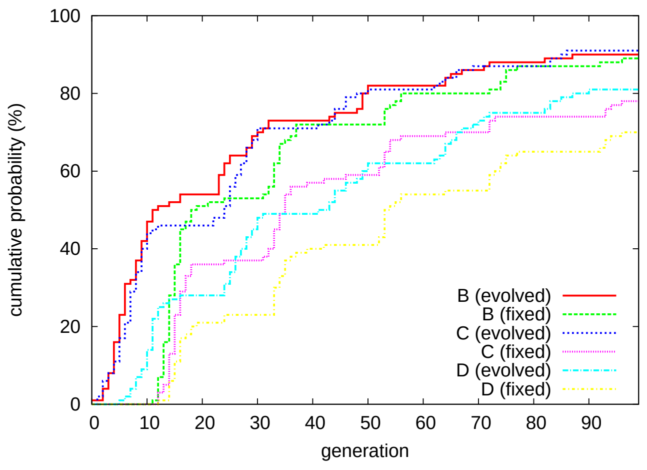

Figure 8: Cumulative distribution functions representing the probabilities that genotypes B, C, and D have evolved and are fixed by the given generation.

Conclusion

We introduced a new type of self-replicating program which (unlike previous self-replicating programs) includes distinct phenotype and genotype components. Although the program is encoded in machine language, and (for this reason) can be executed on a CPU (or reified as a distributed virtual machine) it reproduces by compiling itself from its own source code, which is written in a more expressive high-level language. Because compiling is an intrinsically more complex process than copying, there is a much larger space of implementations to be explored by an evolutionary process; because its genotype is encoded in a high-level language, the space of neighboring self-replicating programs can be more efficiently probed.

To address the problem of how a complicated lexically scoped program like a compiler can evolve into a more complex and efficient program without breaking, we designed, implemented and tested a novel genetic programming system, which uses a pair of mutation operators analogous to gene duplication and specialization, together with homologous crossover and an alternating fitness function which selects for complexity or efficiency depending on epoch. Using this system, we experimentally demonstrated the evolution of several self-replicating programs of increased complexity and efficiency from a less complex and less efficient ancestor. We were able to show that in a population of 200 individuals, the most complex and efficient self-replicating program evolved within 100 generations in over three quarters of all trials, and by crossover of less complex and less efficient parent programs a significant fraction of the time.

Acknowledgements

Thanks to Jeff Barnett, Stephanie Forrest, Ben Edwards, Neal Holtschulte and Melanie Moses for helpful comments.

References

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.

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.

D'haeseleer, P. (1994). Context preserving crossover in genetic programming. In *IEEE World Congress on Computational Intelligence*, pages 27–29.

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

Finnigan, G., Hanson-Smith, V., Stevens, T., and Thornton, J. W. (2012). Evolution of increased complexity in a molecular machine. *Nature*, 481(7381):360–364.

Kirshenbaum, E. (2000). Genetic programming with statically scoped local variables. In *Genetic and Evolutionary Computation (GECCO)*.

Koza, J. (1994). Artificial life: spontaneous emergence of self-replicating and evolutionary self-improving computer programs. In Langdon, C., editor, *Artificial Life III*, pages 225–262. Addison Wesley.

Koza, J., Bennet, F., Andre, D., and Keene, M. (1999). The design of analog circuits by means of genetic programming. *Evolutionary Design by Computers*, pages 365–385.

Offman, M. N., Tournier, A. L., and Bates, P. A. (2008). Alternating evolutionary pressure in a genetic algorithm facilitates protein model selection. *BMC Structural Biology*, 8(34).

Pujol, J. C. F. (1999). *Evolution of artificial neural networks using a two-dimensional representation*. PhD thesis, School of Computer Science, University of Birmingham, UK.

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.

Stephenson, M., Amarasinghe, S. P., Martin, M. C., and O'Reilly, U. (2003). Meta optimization: improving compiler heuristics with machine learning. In Cytron, R. and Gupta, R., editors, *PLDI*, pages 77–90. ACM.

Taylor, T. and Hallam, J. (1997). Studying evolution with self-replicating computer programs. In *Fourth European Conf. on Artificial Life*, pages 550–559. MIT Press.

Williams, L. (2012). Robust evaluation of expressions by distributed virtual machines. In *Unconventional Computation and Natural Computation (UCNC)*, Orleans, France.

Zou, R. and Lung, W. (2004). Robust water quality model calibration using an alternating fitness genetic algorithm. *J. Water Resource Planning Management*, 130(6):471–479.