Autoconstructive Evolution: Push, PushGP, and Pushpop

Lee Spector
Cognitive Science
Hampshire College
Amherst, MA 01002
lspector@hampshire.edu

To appear in *GECCO-2001: Proceedings of the Genetic and Evolutionary Computation Conference*, edited by L. Spector, E. Goodman, A. Wu, W.B. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. Garzon, and E. Burke. Morgan Kaufmann Publishers. 2001.

Autoconstructive Evolution: Push, PushGP, and Pushpop

Lee Spector
Cognitive Science
Hampshire College
Amherst, MA 01002
lspector@hampshire.edu

Abstract

This paper is a preliminary report on autoconstructive evolution, a framework for evolutionary computation in which the machinery of reproduction and diversification (and thereby the machinery of evolution) evolves within the individuals of an evolving population of problem solvers. Autoconstructive evolution is illustrated with Pushpop, an evolving population of programs expressed in the Push programming language. The Push programming language can also be used in a more traditional genetic programming framework and may have unique benefits when so employed; the PushGP system, which uses traditional genetic programming techniques to evolve Push programs, is also described.

1 INTRODUCTION

Autoconstructive evolution is a framework for evolutionary computation in which the machinery of reproduction and diversification (and thereby the machinery of evolution) evolves within the individuals of an evolving population of problem solvers. In this paper autoconstructive evolution is introduced and illustrated with Pushpop, an evolving population of programs expressed in the Push programming language.

The paper begins with a section describing the motivation and historical precedents for autoconstructive evolution. It then introduces Push, a new stack-based programming language that can also be useful in a traditional genetic programming framework. Push supports multiple data types without imposing syntax restrictions and it allows for arbitrary inter-mixing of operations on objects of different types. A `code` data type permits explicit code manipulation that can be used to implement subroutines, macros, and recursion, all without syntax restrictions. Examples of Push programs are presented below, including Push programs evolved within a traditional-style genetic programming system called PushGP (which is also described).

The paper then describes Pushpop, an autoconstructive evolution system built on top of Push. The evolutionary dynamics of Pushpop are complex, and although Pushpop can be used to solve computational problems it often reaches sub-optimal equilibria. Analysis of this behavior engages issues raised in the literatures of evolutionary biology and artificial life. The paper concludes with prospects for the use of PushGP, Pushpop, and autoconstructive evolution in general as problem solving technologies.

2 AUTOCONSTRUCTIVE EVOLUTION

In traditional genetic algorithms (Holland, 1992) and genetic programming systems (Koza, 1992) the mechanisms of reproduction and diversification are designed by human programmers and do not themselves evolve as the systems run. Fit individuals are reproduced, mutated, and recombined according to pre-specified algorithms and with pre-specified parameters such as mutation rates. These algorithms and parameters may or may not be well suited to their tasks, and their effectiveness depends on the ways in which individuals are represented and on the fitness landscapes of the problems to which the systems are being applied.

In contrast, natural evolution presumably began without the imposition of pre-specified mechanisms for reproduction and diversification. These mechanisms presumably co-evolved with the mechanisms underlying all of the other functions of the earliest organisms, including those for metabolizing energy sources. Through this co-evolution the mechanisms for reproduction and diversification adapted to the materials and energy sources of the early Earth, giving rise to the robust evolutionary process that produced our biosphere (Margulis et al., 2000).

Is it possible to follow nature's lead and to produce evolutionary computation systems that adaptively construct their own mechanisms of reproduction and diversification, and thereby their mechanisms of evolution, as they run? There are at least two motivations for exploring the development of such *autoconstructive evolution* systems. The first is that such a system might eventually reach a state in which its evolutionary mechanisms are better suited to a particular problem than any that might be constructed by hand. That is, we can hypothesize that autoconstructive evolution systems will eventually out-perform traditional evolutionary computation systems by adapting their reproductive mechanisms to their representations and problem environments. The second motivation is that such systems may exhibit properties similar to those of biological evolutionary systems, providing useful data for research in evolutionary biology and artificial life.

Several researchers have previously explored the use of adaptive or self-adaptive genetic operators in genetic algorithms, genetic programming, and other forms of evolutionary computation such as "evolution strategies" (Angeline, 1995a, 1996; Bäck, 1992; Stephens, 1998; Hart, 2000). In most previous work, however, the algorithms for reproduction and diversification have been essentially fixed, with only the numerical parameters (such as mutation rates) subject to adaptation. A more radical approach was taken by Edmonds in his "Meta-Genetic Programming" framework, in which the genetic operators (e.g. reproduction, mutation, and crossover) that act on the main population are themselves evolved in an independent, co-evolving population (Edmonds, 1998). Edmonds noted that there is a potential regress here, as genetic operators are required for the evolution of his evolved operators, and he proposed several approaches to this problem including recursive strategies in which operators act on the populations in which they are evolved. Edmonds reported mixed success with his system and described problems with the maintenance of diversity and sensitivity to the details of the special-purpose code-manipulation functions used in genetic operators. Teller had earlier reported work on similar ideas but in a less conventional, graph-based programming framework (Teller, 1994). More recently, Kantschik and colleagues have extended Teller's ideas and reported success with a meta-genetic programming system operating on graph-based programs (Kantschik et al., 1999).

The work described in this paper aims to take one step closer to the natural model, requiring that evolving individuals themselves be responsible for the production of their own children. Just as natural organisms are responsible both for "making a living" in the world (e.g., acquiring and metabolizing food) and for producing children, the individuals in an autoconstructive evolution system are responsible both for solving a problem and for producing new programs for the following generation. The problem-solving and child-producing components of the individuals may be integrated and interdependent, and they may use arbitrary computational processes built in an expressive, Turing-complete programming language.

Autoconstructive evolution has precedents in the artificial life literature, most specifically in the Tierra and Avida systems. In Tierra, programs expressed in an assembly-like language compete for execution time and evolve on a simulated MIMD parallel computer (Ray, 1991). The evolving "creatures" in Tierra are indeed responsible for their own reproduction and their reproductive mechanisms can and do adapt over evolutionary time, but Tierra must be seeded with a hand-coded replicator and it also relies on externally specified mutation mechanisms for evolutionary progress. Further, the creatures in Tierra do not evolve to produce answers to computational problems; while it is useful for the study of evolution it is not a problem-solving technology. The Avida system extends the ideas of Tierra in several ways and it incorporates a mechanism for guiding evolution to solve problems (Adami and Brown, 1995). This mechanism, however, has been applied only to simple problems (such as summing two numbers) and it requires that the user run the system in stages of increasing task difficulty. Like Tierra, Avida requires hand seeding and externally specified mutation mechanisms, and it provides no mechanisms for sexual recombination (e.g. crossover). Nonetheless, Avida exhibits interesting properties akin to those of natural living systems, and it has served as the basis for inter-disciplinary studies of genome complexity and genetic interactions (Lenski et al, 1999).[1]

The artificial life literature contains additional related work. For example, Dittrich and Banzhaf have shown that "algorithmic reaction systems" (sometimes also called "artificial chemistries") are capable of *self-evolution* in which "the components responsible for the evolutionary behavior are (only) the individuals of the population system itself" (Dittrich and Banzhaf, 1998, p. 204). The Dittrich and Banzhaf system exhibits complex evolutionary dynamics without hand seeding

[1] Another recent related system is SeMar (Suzuki, 2000).

but, like Tierra, is not designed to solve computational problems; indeed one of the goals of the project was to produce evolutionary dynamics without any explicit fitness functions or artificial selection procedures.

Many studies have been conducted on general principles of self-replication, beginning with von Neumann's work in the 1940's ((Sipper 1998) contains a survey). Koza has used genetic programming to evolve self-replicating computer programs, but the replication processes that he evolved were independent of the reproductive mechanisms of the genetic programming system (which were hand coded) (Koza, 1992). Few of the prior studies consider in detail the issues involved in simultaneously evolving self-replication and additional problem-solving functionality.

The particular approach to autoconstructive evolution described in this paper relies upon the use of a programming language for the evolving programs with certain unusual properties. Push, the programming language developed to fill this need, can also be used in a more traditional genetic programming framework and may provide independent benefits when so employed. For example it may ease the evolution of programs that use multiple data types, automatically defined functions, and recursion. The next section introduces the Push programming language and provides some examples of its use. This is followed by a section describing PushGP, a traditional-style genetic programming system that evolves Push programs. The paper then returns to autoconstructive evolution and describes Pushpop, an autoconstructive population of Push programs.

3 PUSH

The Push programming language was designed to have a completely uniform syntax (to simplify the evolution of code-manipulating code) while nonetheless supporting multiple data types and high-level control structures such as recursion and named subroutines. It achieves these goals by extending the concepts of stack-based programming languages such as Forth (Salman, 1984); such languages have played a role in previous genetic programming systems and a survey is included in (Bruce, 1997). This section presents the major features of the Push language and provides some illustrative examples, but space limitations do not permit a complete description of the language to be presented here (see (Spector and Robinson, Forthcoming)).

A Push program is a string of instructions, constants, and parentheses, with the only syntax rule being that parentheses must be balanced. As in other stack-based languages, instructions take their arguments from global stacks (one of which exists for each data type) and place their outputs on global stacks. The notation is a form of postfix; for example the following code computes "2 + 3":

```
(integer 2 3 +)
```

When the constant `2` is executed its value is pushed onto the integer stack. Similarly for the constant `3`. The `+` instruction takes two arguments from the integer stack, sums them, and pushes the result (`5`) back onto the integer stack. Like all Push instructions, the + instruction is simply skipped if there are insufficient arguments on the stack when it is executed. The initial `integer` in this program is a type constant. `type` is a built-in data type in Push and therefore has its own stack; when `integer` is executed it is pushed onto the type stack. The `+` instruction, like most Push instructions, consults (but does not pop) the type stack to determine which data type's `+` method to execute. The built-in `float` data type also has a `+` method, and operations on integers and floats can be combined in a variety of ways; all of the following produce identical results:

```
(integer 2 3 + float 2.72 3.14 +)
(2 3 2.72 3.14 integer + float +)
(2.72 integer 2 3.14 3 + float +)
```

The `type` stack has a pre-defined "bottom" that provides type defaults when the executing program provides no appropriate type constants (or when it pops all appropriate type constants that it does provide). The current type stack bottom, listed from top to bottom, is `integer`, `boolean`, `code`, `type`, `name`. If an instruction that is being executed is not implemented for the type on the top of the type stack then the interpreter searches down through the type stack until it finds a type for which the instruction is implemented. Because the pre-defined `type` stack bottom must include at least one type that implements each instruction, an appropriate type will always be found.

Generic stack manipulation instructions such as `pop` (remove the top element of a stack), `dup` (push a duplicate of the top element), and `swap` (swap the top two elements) are implemented for all data types. A `convert` instruction converts between the two topmost types on the `type` stack; `convert` methods are provided for all pairs of the data types installed in the system (for example, floats are converted to integers by truncation).

Parentheses generally have no effect on execution, so for example `(+ 2 3)`, `(+ (2 3))`, and `((+) 2`

((3))) all produce identical results. Parentheses serve only to group and provide hierarchical structure to code, which can be of great utility when code is manipulated as data (and possibly later executed). Code manipulation is accomplished via the code data type, which inherits functionality from the expression data type. A rich set of list-manipulation instructions, inspired by those that form the core of the Lisp programming language (Graham, 1996), allows for arbitrary symbolic manipulation of expressions. The quote instruction is an essential ingredient of code-manipulation expressions and it is handled as a special case by the interpreter. When quote is executed a flag is set in the interpreter that will cause the next piece of code submitted for execution (which may be a parenthesized sub-program) to be pushed onto the most current expression stack rather than being executed. The do instruction recursively invokes the interpreter on the expression on the top of the code stack (while leaving its code argument on the stack until the recursive execution is complete; do* pops the code argument first). So for example the following is a complicated way to add 2 and 3:

```
(code quote (integer 2 3 +) do)
```

When code is submitted to the top level interpreter for execution it is pushed on the code stack prior to execution. This provides a convenient way to write recursive code, such as the following code which recursively computes the factorial of an input provided on the integer stack:

```
(code quote
       (quote (pop 1)
        quote (code dup integer dup 1 - do *)
        integer dup 2 < if)
      do)
```

This code also uses the if instruction, which executes one of the top two items on the code stack (and discards the other), depending on what is found on the top of the boolean stack. In this case the item on top of the boolean stack will have been left there by the integer method for the < instruction, which will have compared the input to 2.

Note that the code manipulation features of Push make it easy to write self-modifying programs. Such programs might have "morphological" phases during which they develop into "mature" code which is then executed to solve a problem. Alternatively, such programs might continue to develop as they run, exhibiting "ontogeny" more in the manner of living organisms. Prior work on morphogenic evolutionary computation and on ontogenetic programming required unusual mechanisms to achieve these effects (Angeline, 1995b; Spector and Stoffel, 1996a, 1996b), whereas systems based on Push may get them "for free" via the code manipulation features of the language.

The recursive execution capabilities of do, do*, if, and map (which is an iterator similar to Lisp's mapcar) introduce the possibility of nontermination. For this reason one can specify an upper bound on the number of instructions that can be executed for a single top-level call of the interpreter. If this bound is exceeded then execution terminates immediately and the calling program can either use or discard the "results" that have been left on the stacks at that time.

A name data type, in conjunction with set and get instructions, provides variables that can be used to hold items of any type (including code for named subroutines). Any symbol that is not an instruction name or constant of any other sort (including type constants) is considered a name constant and is pushed on the name stack for use with subsequent calls to set or get. For example, the following is a version of the factorial program with a named subroutine:

```
(code factorial
      quote
        (code
         quote (pop 1)
         quote (dup 1 - code factorial get do *)
         integer dup 2 < if)
      set do)
```

Figure 1 shows the Push type hierarchy and some of the currently implemented instructions for each type.

4 PUSHGP

Push is a flexible language that can be used either in an autoconstructive evolution framework or in a traditional genetic programming framework. This section describes PushGP, a traditional genetic programming system that evolves Push programs.

The basic algorithm of PushGP is the same as that for standard genetic programming. PushGP begins by generating a population of random programs.[2] Each of these is evaluated for fitness. If a sufficiently fit program is found it is printed and the system halts. Otherwise a new generation of programs is produced by applying reproduction, mutation, and crossover operators to the more-fit programs in the current generation. PushGP uses tournament selection and reasonably standard genetic operators; mutation replaces

[2]In principle any of the random program generation techniques used for genetic programming could be trivially adapted for PushGP. For details on the techniques used in PushGP see (Spector and Robinson, Forthcoming).

```
- push-base-type:
      dup, pop, swap, rep, =[boolean], set[name],
      get[name], convert[type], pull[integer],
      noop
  - number: +, -, *, /, >[boolean], <[boolean]
    - integer: rand, pull, /
    - float: rand
  - boolean: not, and, or, nand, nor, rand
  - expression:
      quote, car, cdr, cons, list, append, subst,
      container, length[integer], size[integer],
      atom[boolean], null[boolean], nth[integer],
      nthcdr[integer], member[boolean],
      position[integer], contains[boolean],
      insert[integer], extract[integer],
      instructions[type], perturb[integer],
      other[integer], other-tag[float],
      elder[integer], neighbor[integer],
      rand[integer]
    - code: do, do*, if[boolean], map
    - child:
  - type: rand
  - name: rand
```

Figure 1: The Push type hierarchy and some of the currently implemented instructions for each type. A type name in square brackets means that the preceding instruction accesses the named stack in addition to the stack of the type to which the instruction is being applied. Sub-types inherit instructions from their super-types, though they may override the implementation of these instructions with more specific methods. The type hierarchy can be easily extended. Some of the runs described in this paper use a subset of the type hierarchy or a subset of the instructions shown here.

a randomly chosen sub-expression with a new random subexpression, and crossover replaces a randomly chosen sub-expression with a randomly chosen sub-expression from another individual.[3] The process continues until a solution is found or until the maximum number of generations has been exceeded.

4.1 AN ODD SOLUTION TO THE *ODD* PROBLEM

One of the strengths of the Push language is that it allows for the flexible integration of code that manipulates data of different types. The more traditional approach to the evolution of programs with multiple data types, *strongly typed genetic programming,* imposes syntax restrictions to ensure that functions are passed data only of the types that they require (Montana, 1995). This complicates the construction (or evolution) of genetic operators and has other impacts on evolution, for example by limiting options for crossover. Push allows for an approach free from such complications. Whether or not this will ultimately be advantageous, in terms of the computational effort required to evolve large, multiple-type programs, is an open, empirical question that is not addressed in this paper. Here we aim only to illustrate the approach and some of its possibilities.

As a rudimentary demonstration of its multiple-type capabilities PushGP was given the task of evolving a program for the *odd* problem. In this problem the program receives as input a single `integer` and is required to produce as output a `boolean` value indicating whether the input is an odd number; the answer should be `T` if the input is odd and `NIL` otherwise.[4]

PushGP was run on this problem with a population of 1000, tournament size 5, and genetic operator rates of 40% crossover, 40% mutation, and 20% straight reproduction. The maximum size for programs was 100 points, initial random programs were limited to 15 points, and the execution step limit was 100.[5] Each program was evaluated for fitness on the twenty integers from 0 to 19. In the fifth generation the following correct solution was produced:

```
((nth) atom (insert) pull)
```

This is a most unusual solution that uses no arithmetic instructions, even though the language's full complement of arithmetic instructions were available. This solution can be understood by considering the following:

- The `nth` instruction takes an expression and an integer and pushes onto the appropriate expression stack the element of the expression indexed by the integer.[6]
- In this case the appropriate expression stack is the

[3]For the experiments described in this paper nodes were chosen for mutation and crossover using uniform random selection. A more recent version of the system allows for node selection to be biased in favor of internal nodes (as opposed to leaves), as is common practice in genetic programming.

[4]The current version of Push is implemented in Lisp and therefore inherits Lisp's conventions for the representation of true and false (Graham, 1996).

[5]Each instruction, constant, or pair of parentheses counts as one point. These parameters were chosen arbitrarily; there is no particular justification for their values and they were not optimized. A study of the effects of different values for these parameters is currently in progress.

[6]If the item on top of the appropriate expression stack is just an atom then a list containing just that atom is used instead.

code stack, as specified in the type stack bottom.

- nth is zero-based; the first item in the expression is item 0, the second is item 1, and so on.
- nth "wraps around" to the beginning of the expression if its integer input is greater than the length of the expression.
- The atom instruction pushes NIL (meaning "false") onto the boolean stack if its input is surrounded by parentheses, and T (meaning "true") otherwise.

Recall that the program starts with its integer input on the integer stack. The execution of the nth instruction accesses the program's *own code* and pushes onto the top of the code stack the component of the program indexed by the integer input. Note that this component will be an "atom" (that is, not surrounded by parentheses) if and only if the input number is an odd number. The next instruction, atom, will therefore leave T on top of the boolean stack if the input number is odd, and NIL on top of the boolean stack otherwise, correctly solving the problem. The remaining two instructions have no effect on the boolean stack and can therefore be ignored, aside from noting that they maintain the pattern of alternating atoms upon which the program relies. For this reason the program can be simplified to: ((nth) atom).

This remarkably concise program, which uses its own code as an auxiliary data structure, nicely illustrates that multiple data types can sometimes be used in unexpected ways, providing synergistic advantages.

4.2 PARITY AND MODULARITY

Another strength of the Push language is that it permits the expression, without syntax restrictions, of modules such as functions and macros. The more traditional approaches to the evolution of programs with subroutines (*automatically defined functions,* or ADFs (Koza, 1994)) and macros (*automatically defined macros,* or ADMs (Spector, 1996)) require that one specify in advance how many such modules are to be used and how many arguments they will receive. Syntax restrictions are also imposed by the definitions of the modules, complicating the expression of genetic operators and in some cases limiting the possibilities of crossover. A more refined approach using *architecture altering operations* eliminates the need for prespecification of the numbers of modules and arguments, but it does so at the expense of additional complexity (Koza et al., 1999). Similarly to the case of multiple-data types described above, Push allows for an approach free from such complications. Again, whether or not this will ultimately be advantageous, in terms of the computational effort required to evolve large, modular programs, is an open, empirical question that is not addressed in this paper (but see (Spector and Robinson, Forthcoming)). Here we aim only to illustrate the approach and some of its possibilities.

Parity problems have been used widely to test and demonstrate genetic programming systems, and in particular they have been used to demonstrate the utility of modularity (Koza, 1994). As a rudimentary demonstration of its ability to automatically evolve novel control structures, PushGP was given the task of evolving a program for the *even 4-parity* problem. The input for this problem consists of four Boolean values, and the output, also a Boolean value, should be T (true) if the number of T inputs is even (and NIL otherwise). For this run we eliminated numerical types and limited the Boolean instructions to and, or, nand, and nor (to conform to the function set used in (Koza, 1994)).

PushGP was run on this problem with a population of 10000, tournament size 5, and genetic operator rates of 40% crossover, 40% mutation, and 20% straight reproduction. The maximum size for programs was 50 points, initial random programs were limited to 25 points, and the execution step limit was 200.[7] Each program was evaluated for fitness on all 16 possible combinations of 4 Boolean inputs. In the 71st generation a correct 45-point solution was produced, a simplified (41-point) version of which follows:

```
(quote (x x (x ((x) x)))
 (list (x) ((x) (x quote (dup nand) if) nil)
  (x x) ((quote) ((x) x x) x (map nor))))
```

In this version the name constant x was substituted for instructions that have no effect on the program's output, but which cannot be removed because they play a structural role in the program's execution. As with the *odd* program above, this program uses its own code as an auxiliary data structure, but in this case the mode of operation of the program is not so clear. Indeed it is quite difficult to explain how this program works, and a full explanation will not be included here.

Is this solution modular? In some respects it clearly is. For example, it is recursive (calling itself via map, an instruction that iteratively executes a body of code on each element of a list of expressions). Further, although the text of the program contains only one instance of the nor function, each run of the program

[7]Again, these parameters were chosen arbitrarily and were not optimized.

executes `nor` four times. The same holds for `list`, `map`, and `if`. `nand` also occurs only once in the text of the program but is executed 1, 2, 3, or 4 times depending on the input to the program. The same is true of `dup`. So the program is clearly re-using code, which is one of the hallmarks of modularity. On the other hand, this is probably not the sort of modularity that any human would employ.

5 PUSHPOP

5.1 PRODUCTION OF CHILDREN

Pushpop can perhaps best be understood by beginning with PushGP and specifying that individual programs will be responsible for the production of their own children. One way to do this would be to declare that whatever is left on top of the `code` stack at the end of a program's execution should count as a potential child. Because a program is typically executed several times for each fitness evaluation (once for each input or "fitness case") this would produce several potential children for each program from each fitness evaluation. This is essentially the strategy used in Pushpop, with the refinement that a dedicated `expression` type (with a stack), called `child`, is provided for the purpose of child production. This prevents complications that otherwise might occur from using the same `code` stack both for child production and for other purposes (such as recursion).

5.2 SELECTION

Since children are produced by the individual programs as they are evaluated for fitness, there is no role for the standard genetic operators of mutation, crossover, and reproduction. But something must come in their stead, not to produce but to *filter* the children that will be admitted to the next generation. This is necessary both to keep the population size limited and to provide selective pressure; only by allowing more of the children of the better parents to survive does Pushpop encourage the development of fitter programs. The filtering is accomplished via tournaments: for each position in the next generation n random individuals are picked from the current generation, and a random child of the best of these n individuals is chosen to survive. The tournament size n is an environmental constant (currently 2) that can be used to adjust the selection pressure.

5.3 SEX

Sexual recombination, involving any number of individuals and any method of code recombination, can be expressed in Pushpop as long as there is some mechanism that allows the code of one individual to refer to the code of other individuals. The current version of Push provides four instructions with this capability, each of which pushes the code of another program onto the current `expression` stack:

- `neighbor`: Takes an integer n and returns the code of the individual distance n in the population from the current individual. The population has a linear structure with siblings grouped together; low n will likely return a close relative and high n will likely return an unrelated program.
- `elder`: Takes an integer that is used as a tournament size for a tournament among the individuals of the *previous* generation. The program of the winning individual is returned.
- `other`: Takes an integer tournament size and performs a tournament among individuals of the *current* generation, comparing individuals with respect to their *parents'* fitnesses (since their own fitnesses will in general not yet be known).
- `other-tag`: Takes a floating-point *tag* and searches for a program in the current generation containing the tag. This is slow and is therefore usually disabled. Programs can achieve a similar effect through combinations of `neighbor` and expression comparison instructions.

5.4 DIVERSITY MANAGEMENT

As noted by Edmonds in his prior work (Edmonds, 1998), it can be difficult to maintain diversity in a system with evolving genetic operators. Perfect replication operators, for example, can quickly homogenize a population, as can other operators that insufficiently diversify their outputs. Because we have limited resources (particularly compared to those that were available for the one known prior instance of completely autoconstructive evolution: the evolution of life on Earth) and must therefore limit ourselves to relatively small populations, we must take strong measures to ensure that the population remains sufficiently diverse. Pushpop maintains diversity through a combination of syntactic and semantic diversity constraints. The syntactic diversity constraints specify that children cannot be identical to their parents and that a population can never contain any completely identical programs. Semantic diversity can be encouraged through a variety of mechanisms. One form of semantic diversity constraint built into Pushpop limits the number of children that can be produced by parents with a particular fitness value (the limit is a system parameter called "semantic niche size"). In another scheme the weighting of the various fitness cases is varied across the population, creating different ecological pressures in different "geographical" areas and thereby discouraging semantic homogeneity.

5.5 REPRODUCTIVE COMPETENCE

At the beginning of a run one must first achieve diversifying recursive replication; in other words one must obtain individuals that make diverse children that are themselves capable of making diverse children (which can in turn make diverse children, etc.). Once such individuals arise their descendents will fill the population; selection will then come into play, favoring the survival of the children of the more-fit parents. Until such reproductive competence has been achieved there will be too few "naturally born" children to fill the population, so the population is padded with newly generated random individuals. In order to speed the achievement of reproductive competence one can fortify the primordial soup by increasing the probability that instructions and constants useful for diversifying reproduction (such as `child`, `neighbor`, and `rand`) are included in random programs. Note that this is quite different from the way in which systems like Tierra are seeded with hand-coded replicators, since none of the instructions in the fortified soup is reproductively competent on its own.

5.6 RESULTS AND REFINEMENTS

Pushpop has been run on a range of symbolic regression problems (Koza, 1992), with a range of values for environmental parameters (including population size, instruction set, tournament size, and semantic niche size). Under most conditions the population quickly achieves reproductive competence and soon thereafter improves in fitness. Often there are a number of subsequent improvements in fitness, and sometimes a completely correct solution is found. More often than not, however, a stable (though diverse) equilibrium is reached before the target problem is solved.

This should not come as a great surprise, as many studies have questioned the "progressiveness" of the natural evolutionary processes that autoconstructive evolution aims to emulate. A recent study by Turney suggests that evolution can indeed exhibit global progressive trends and that the primary trend is toward increasing "evolutionary versatility" (Turney, 2000). Turney's computational experiments further imply that a key requirement for unbounded evolutionary versatility is a highly dynamic fitness landscape. This accords well with arguments in evolutionary biology about the role of climate change in evolutionary progression, and suggests a straightforward refinement to Pushpop that may encourage continued progress. This refinement, *fitness case rotation*, involves gradually changing the set of fitness cases used for fitness evaluation from generation to generation. In the simplest case with an even number n of fitness cases one can use cases 1 through $\frac{n}{2}$ in the first generation, cases 2 through $\frac{n}{2}+1$ in the second generation, etc., wrapping around to case 1 after case n. This has been implemented in Pushpop and while the results are still preliminary it seems to have a beneficial effect in promoting continued fitness improvement.

Other ideas from evolutionary biology may also be helpful. For example Turney notes that Gould explains apparent evolutionary progress as the result of random variation coupled with a minimum level of complexity required for life. This notion has been incorporated into Pushpop as an optional requirement that an individual's fitness must be better than some minimum value (for example the fitness of a null program) for its children to survive. Further study is required to determine the effect of this restriction.

The complex evolutionary dynamics of Pushpop provide a rich source of data for other connections to evolutionary biology. For example, a symbolic regression run with the target equation $y = x^3 + x^2 + x$ produced the following correct solution:

```
(size perturb < integer convert convert dup dup *
rand subst (extract) (subst) (+ code *) ((pop)
(set t child +) (convert convert) (null 0.011))
nil * set erc-var-193802 /)
```

This solution scores perfectly on the fitness test but reproduces asexually. Further analysis shows that there were large spikes in the usage of instructions required for sexual recombination (`elder`, `other`, and `neighbor`) shortly after the population reached reproductive competence (at generation 75) and through the subsequent period of major fitness improvement. By generation 550, however, when the solution above was produced, these instructions were being called only about once every three program executions (down from a high of about four such calls per program execution). It appears that sexual reproduction played a major role in the initial improvement of the population but that later generations relied on incremental asexual strategies based on the `perturb` instruction.[8] Further study of the adaptation of reproductive strategies in Pushpop and their relation to adaptations in biological reproduction may eventually provide insights that will improve Pushpop and/or shed light on biological evolutionary processes.

6 CONCLUSIONS

This is a preliminary report on a recently developed line of research and the conclusions are therefore tentative. The Push programming language has been shown to have several features that can support novel evolutionary computation systems. PushGP, a traditional-style genetic programming system that evolves Push programs, may offer advantages in the evolution of complex programs using multiple data types and control structures; further study is required to assess its strengths and weaknesses. Pushpop is an autoconstructive evolution system consisting of Push programs that construct their own children while solving computational problems. Further study is required to determine if Pushpop can fulfill the hypothesized promise of autoconstructive evolution systems to out-perform traditional evolutionary computation systems by adapting their reproductive mechanisms to their representations and problem environments (Spector and Robinson, Forthcoming).

Acknowledgments

Mark Feinstein, Raymond Coppinger, Wolfgang Banzhaf, Sean Luke, Alan Robinson, Chris Perry, Jaime Davila, Christopher Jarvis, Herbert J. Bernstein, Rebecca S. Neimark, and anonymous reviewers provided many valuable comments that helped to improve this work.

This effort was sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30502-00-2-0611. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory, or the U.S. Government.

This research was also made possible by generous funding from Hampshire College to the Institute for Computational Intelligence at Hampshire College.

[8]`perturb` stochastically replaces atoms in an expression with random new atoms according to probabilities specified by an integer argument.

References

Adami, C., and C.T. Brown. 1995. Evolutionary Learning in the 2D Artificial Life System "Avida". In *Artificial Life IV.* MIT Press, pp. 377-381.

Angeline, P.J. 1995a. Adaptive and Self-Adaptive Evolutionary Computations. In *Computational Intelligence: A Dynamic Systems Perspective.* IEEE Press, pp. 152-163.

Angeline, P.J. 1995b. Morphogenic Evolutionary Computations: Introduction Issues and Examples. In *Evolutionary Programming IV: The Fourth Annual Conference on Evolutionary Programming.* MIT Press, pp. 387-401.

Angeline, P.J. 1996. Two Self-Adaptive Crossover Operators for Genetic Programming. In *Advances in Genetic Programming 2.* MIT Press, pp. 89-110.

Bäck, Thomas. 1992. Self-Adaptation in Genetic Algorithms. In *Toward a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life.* MIT Press, pp. 263-271.

Bruce, W.S. 1997. The Lawnmower Problem Revisited: Stack-Based Genetic Programming and Automatically Defined Functions. *Genetic Programming 1997: Proceedings of the Second Annual Conference.* Morgan Kaufmann, pp. 52-57.

Dittrich, P., and W. Banzhaf. 1998. Self-Evolution in a Constructive Binary String System. *Artificial Life*, Vol. 4, pp. 203-220.

Edmonds, B. 1998. Meta-Genetic Programming: Co-evolving the Operators of Variation. CPM Report No.: 98-32. Centre for Policy Modelling, Manchester Metropolitan University. http://www.cpm.mmu.ac.uk/cpmrep32.html

Graham, P. 1996. *ANSI Common Lisp.* Prentice Hall.

Hart, W.E. 2000. A Convergence Analysis of Unconstrained and Bound Constrained Evolutionary Pattern Search. *Evolutionary Computation,* 9(1), pp. 1-23.

Holland, J.H. 1992. *Adaptation in Natural and Artificial Systems.* MIT Press.

Kantschik, W., P. Dittrich, M. Brameier, and W. Banzhaf. 1999. MetaEvolution in Graph GP. *Proceedings of EuroGP'99*, LNCS, Vol. 1598. Springer-Verlag, pp. 15-28.

Koza, J.R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press.

Koza, J.R. 1994. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press.

Koza, J.R., D. Andre, F. H Bennett III, and M. Keane. 1999. *Genetic Programming 3: Darwinian Invention and Problem Solving.* Morgan Kaufmann.

Lenski, R.E., C. Ofria, T.C. Collier, and C. Adami. 1999. Genome Complexity, Robustness and Genetic Interactions in Digital Organisms. *Nature* Vol. 400, pp. 661-664.

Margulis, L., D. Sagan, and N. Eldredge. 2000. *What is Life?* University of California Press.

Montana, D.J. 1995. Strongly Typed Genetic Programming. *Evolutionary Computation,* 3(2), pp. 199-230.

Ray, T.S. 1991. Is it Alive or is it GA? In *Proceedings of the Fourth International Conference on Genetic Algorithms.* Morgan Kaufmann, pp. 527-534.

Salman, W.P, O. Tisserand, and B. Toulot. 1984. *FORTH.* Springer-Verlag.

Sipper, M. 1998. Fifty Years of Research on Self-Replication: An Overview. *Artificial Life*, Vol. 4, No. 3, pp. 237-257.

Spector, L. 1996. Simultaneous Evolution of Programs and their Control Structures. In *Advances in Genetic Programming 2*, MIT Press.

Spector, L., and K. Stoffel. 1996a. Ontogenetic Programming. In *Genetic Programming 1996: Proceedings of the First Annual Conference.* MIT Press, pp. 394-399.

Spector, L., and K. Stoffel. 1996b. Automatic Generation of Adaptive Programs. In *From Animals to Animats 4: Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior.* MIT Press, pp. 476-483.

Spector, L., and A. Robinson. Forthcoming. Genetic Programming and Autoconstructive Evolution with Push. In preparation for submission to *Genetic Programming and Evolvable Machines.*

Stephens, C.R., I.G. Olmedo, J.M. Vargas, and H. Waelbroeck. 1998. Self-Adaptation in Evolving Systems. *Artificial Life*, Vol. 4, pp. 183-201.

Suzuki, H. 2000. Evolution of Self-Reproducing Programs in a Core Propelled by Parallel Protein Execution. *Artificial Life*, Vol. 6, No. 2, pp. 103-108.

Teller, A. 1996. Evolving Programmers: The Coevolution of Intelligent Recombination Operators. *Advances in Genetic Programming 2*, MIT Press, 45-68.

Turney, P.D. 2000. A Simple Model of Unbounded Evolutionary Versatility as a Largest-Scale Trend in Organismal Evolution. *Artificial Life*, Vol. 6, No. 2, pp. 109-128.