The Push3 Execution Stack and the Evolution of Control

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

Jon Klein
Cognitive Science, Hampshire College, Amherst, MA, USA,
and Physical Resource Theory, Chalmers U. Tech. & Göteborg U., Göteborg, Sweden
jk@artificial.com

Maarten Keijzer
Chordiant Software Inc.
Amsterdam, The Netherlands
mkeijzer@cs.vu.nl

ABSTRACT

The Push programming language was developed for use in genetic and evolutionary computation systems, as the representation within which evolving programs are expressed. It has been used in the production of several significant results, including results that were awarded a gold medal in the Human Competitive Results competition at GECCO-2004. One of Push's attractive features in this context is its transparent support for the expression and evolution of modular architectures and complex control structures, achieved through explicit code self-manipulation. The latest version of Push, Push3, enhances this feature by permitting explicit manipulation of an execution stack that contains the expressions that are queued for execution in the interpreter. This paper provides a brief introduction to Push and to execution stack manipulation in Push3. It then presents a series of examples in which Push3 was used with a simple genetic programming system (PushGP) to evolve programs with non-trivial control structures.

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming—*program synthesis*; D.3.2 [**Programming Languages**]: Language Classifications—*specialized application languages*

General Terms

Algorithms, Experimentation, Languages

Keywords

Push, stack-based genetic programming, combinators, recursion, iteration, reversing a list, factorial, Fibonacci sequence, parity, exponentiation, sorting

GECCO'05, June 25–29, 2005, Washington, DC, USA.

1. INTRODUCTION

Complex programs written by humans almost always make use of control structures including conditionals, loops, and recursions. Genetic programming researchers have long appreciated this fact and have developed a number of ways in which programs evolved in genetic programming systems can potentially make use of sophisticated control structures. Some of the earliest work allowed for conditionals ("if then else" structures) and some forms of iterative loops ("do until" structures), and a variety of schemes have been proposed to allow for modules (sometimes also called subroutines, defined functions, automatically defined functions, automatically defined macros, or products of encapsulation); see for example [8, 10, 2, 6, 22, 18, 19]. Additional work has explored the use of implicit or explicit recursion in evolved programs (for example [15, 16, 31, 3, 34, 33, 38, 14, 35, 36, 20, 7, 32, 11]).

The Push programming language provides an alternative mechanism for the expression and evolution of arbitrary control structures [23, 28, 24, 27]. It is not possible in a paper of this length to compare the Push approach to all of the approaches mentioned in the previous paragraph, and this paper has no such comparative goal; some comparisons can be found in other publications on Push cited above. The objective of this paper is rather to present new features in Push3 that further facilitate the evolution of novel control structures — primarily the facilities for execution stack manipulation — and to demonstrate through a series of examples the kinds of results that the approach can routinely produce. Some of these results are striking both for their power and for their novelty.

The next section of the paper outlines the Push language to provide context for the innovations in Push3 that are described in the subsequent section. The discussion of the Push3 execution stack is followed by a series of case studies in which Push3 was used with a genetic programming system (PushGP) to evolve programs that incorporate non-trivial control structures.

2. PUSH

The Push programming language was developed specifically for genetic and evolutionary computation. Among its virtues for such applications is its combination of an unusually simple syntax with the ability to work flexibly with multiple datatypes.

The syntax of a Push program is simply:

```
program ::= instruction | literal | ( program* )
```

That is, a Push program is an instruction, a literal, or a parenthesized sequence of zero or more Push programs. The only syntactic restriction is that parentheses must be balanced. Because Push programs are typically stored and manipulated as tree structures within which the parentheses are implicit, this restriction is usually automatically enforced.

With respect to handling multiple datatypes, Push achieves its flexibility through the use of a stack-based execution architecture with one stack for each type. Genetic programming with Push extends prior work on stack-based genetic programming (e.g. [17, 29, 30]) by providing multiple stacks, one per type. Types are provided for integers, floating point numbers, Boolean values, symbolic names, and code (described in more detail below), each of which has a corresponding data stack. Additional types for vectors, matrices, and other data are provided in some implementations, and it is straightforward to add new types.[1] As instructions are executed they pop any required input values from the appropriate stacks, preform calculations, and push any output values onto the appropriate stacks. The types of the values that will be needed or produced are specified in the implementations of the instructions themselves, and are independent of the syntactic contexts in which calls to the instructions occur. This scheme ensures that Push instructions will always receive inputs and produce outputs of the appropriate types, regardless of the structure of the programs in which they occur. Whenever an instruction finds insufficient items on the stacks for its inputs it acts as a "no-op" and has no effect.

Instructions in Push3 are typically given names such as <TYPE>.<NAME>, where NAME specifies the operation and TYPE specifies the data type upon which the operation should be preformed. INTEGER.=, for example, takes two input values from the INTEGER stack, compares them, and places the result of the comparison on the BOOLEAN stack. It is not uncommon for the same operator to be implemented for multiple types. The instructions INTEGER.POP, FLOAT.POP, and CODE.POP, for example, each pop the top item from the corresponding stack.

The full Push instruction set is large and cannot be fully documented here,[2] but a sample of some of the more commonly used Push instructions is shown in Table 1. The instructions shown on the right-hand side are implemented for each of the types described in the left column, so the instruction MAX, for example, exists both as INTEGER.MAX and as FLOAT.MAX.

Flexibility with respect to control arises because CODE is itself a native type in Push. A Push program can put code on the CODE stack (for example, with the CODE.QUOTE instruction), duplicate or otherwise manipulate it, and later execute it by means of other CODE instructions. This allows programs to dynamically create novel control structures and subroutine architectures. Examples of several such results in earlier versions of Push are described elsewhere [23, 28, 24, 27].

Table 1: Sample Push instructions.

Stack manipulation instructions (all types)	POP, SWAP, YANK, DUP, STACKDEPTH, SHOVE, FLUSH, =
Math (INTEGER and FLOAT)	+, −, /, *, >, <, MIN, MAX
Logic (BOOLEAN)	AND, OR, NOT, FROMINTEGER
Code manipulation (CODE)	QUOTE, CAR, CDR, CONS, INSERT, LENGTH, LIST, MEMBER, NTH, EXTRACT
Control manipulation (CODE and EXEC)	DO*, DO*COUNT, DO*RANGE, DO*TIMES, IF

Code manipulation by evolving programs can also support entirely new forms of evolutionary computation such as "autoconstructive evolution," in which evolving programs must generate their own offspring, eschewing hardcoded genetic operators in favor of evolved genetic operators that are implemented by code-manipulation instructions working on the CODE stack. The results of experiments employing autoconstructive evolution in earlier versions of Push can be found in [23, 25, 26].

3. THE PUSH3 EXEC STACK

3.1 Push Program Interpretation

The most significant change to the Push language in Push3 is the introduction of the EXEC stack, which stores expressions, instructions, and literals that the Push interpreter will subsequently execute. This stack is independent of the CODE stack, which can still be used for code manipulation and for general list manipulation. Code on the CODE stack is static data unless it is executed with an instruction like CODE.DO* or CODE.DO*TIMES; such instructions, which were formerly implemented by (single or multiple) recursive calls to the interpreter, are now implemented by moving code to the EXEC stack. In contrast with the CODE stack, the EXEC stack holds the code that is queued for execution in the interpereter, and it is continuously executed. Although the EXEC stack execution model of Push3 is backward compatible with program execution in Push2, it nonetheless represents a fundamental change in the way that Push programs are executed and it does so in a way that provides new opportunities for the evolution of arbitrary control.

In Push2, programs were executed according to the following algorithm:

- To execute program P:
  1. If P is an INSTRUCTION: execute P (accessing whatever stacks are required).
  2. If P is a LITERAL: push P onto the appropriate stack.
  3. If P is a LIST: recursively execute each subprogram in P.

In this scheme an interpreter that encounters a list must maintain the state of the computation for continuation after

[1] Support for the definition of new types from within Push programs is not part of the current Push3 specification, although several proposals for accomplishing this are under consideration.

[2] See [27].

returning from recursive calls; for example, when executing a list of two subprograms the interpreter must store the second (for later execution) while recursively executing the first. If the Push interpreter is implemented in a language that supports recursion then this can be handled by the language's native mechanisms, which presumably store local variables in activation records during recursive calls. Push3, by contrast, performs the same computation by storing all of the necessary information within the interpreter itself, on an `EXEC` stack:

- To execute program P:
  1. Push P onto the `EXEC` stack.
  2. While the `EXEC` stack is not empty, pop and process the top element of the `EXEC` stack, E:
     (a) If E is an instruction: execute E (accessing whatever stacks are required).
     (b) If E is a literal: push E onto the appropriate stack.
     (c) If E is a list: push each element of E onto the `EXEC` stack, in reverse order.

All of the Push2 control structures (e.g. `CODE.DO*TIMES`) are expressed in Push3 as sequences of instructions that are pushed onto the `EXEC` stack and subsequently executed by the loop in step 2 above. The `CODE.DO*COUNT` instruction, for example, was implemented in Push2 as a loop in the Push interpreter's native language that would repeatedly push counter values on to the `INTEGER` stack and then execute code obtained from the `CODE` stack. In Push3, the `CODE.DO*COUNT` instruction simply pushes code (including a recursive call) and integers onto the `EXEC` stack, and the continued execution of elements from the `EXEC` stack produces the same results.[3] Other features of Push can also be more elegantly implemented in Push3 than in Push2; for example the `CODE.QUOTE` instruction, which formerly required an exception to the standard evaluation rule and a global flag, can now be implemented simply by copying the top of the `EXEC` stack to the `CODE` stack (making it the inverse of `CODE.DO*`).

At first glance the use of the `EXEC` stack does not appear to be a dramatic departure from the program execution algorithm used in Push2. The power of this approach becomes evident, however, when one considers what it means to *manipulate* the `EXEC` stack during a computation. Just as control structures can be implemented by manipulating and later executing items on to the `CODE` stack, novel control structures can also be implemented through `EXEC` stack manipulation and these implementations are often more parsimonious (and therefore potentially more evolvable).

Since a list of code to be executed is placed on the `EXEC` stack in reverse order, `EXEC` instructions have the property of operating on elements in the code which come *after* them, unlike operators applied to other types which use the postfix notation standard in stack-based languages. The following two programs fragments, for example, both produce the same results:

```
( 5 CODE.QUOTE ( INTEGER.+ ) CODE.DO*COUNT )
( 5 EXEC.DO*COUNT ( INTEGER.+ ) )
```

[3]The new `EXEC.DO*COUNT` instruction is equivalent except that it takes its code argument from the `EXEC` stack. Other `EXEC` versions of pre-existing `CODE` instructions are analogous.

3.2 Combinators

The stack manipulation instructions that are provided for all types in Push can be used to manipulate the `EXEC` stack, but the `EXEC` stack can also be manipulated with Push versions of the standard combinators K, S and Y [21, 5]. These combinatory logic operators allow complex computational processes to be built up from simple expressions on the `EXEC` stack.

The combinator `EXEC.K` simply removes (and discards) the second element from the `EXEC` stack. For example, if the `EXEC` stack contains $[A, B, C, ...]$ then executing `EXEC.K` yields $[A, C, ...]$. The combinator `EXEC.S` pops three items, A, B and C from the `EXEC` stack and then pushes back three separate items: `(`B C`)`, C and A (leaving the A on top). Note that this produces two calls to C. The fixed point Y-combinator instruction `EXEC.Y` can also be used to implement recursion using anonymous expressions on the `EXEC` stack; it inspects (but does not pop) the top of the `EXEC` stack, A, and then inserts the list `(EXEC.Y` A`)` as the second item on the `EXEC` stack. By itself, this generates an endlessly recursive call to the unnamed non-recursive "function" A. Recursion can be terminated through further manipulation of the `EXEC` stack that may occur, possibly conditionally, within A.

3.3 Re-entrance

An additional benefit of the `EXEC` stack is that the state of a Push interpreter can now be fully specified by its configuration, its `NAME` bindings, and the contents of its stacks. No internal state variables such as loop counters, execution pointers or continuations are necessary. Among other things, this makes Push interpreters fully re-entrant and allows stricter control over program execution. Loops, previously implemented in the native language's for-loop (or analogous control structure), are now implemented by pushing a series of elements onto the `EXEC` stack. Execution of the loop proceeds through the sequential execution of the elements on the `EXEC` stack.

Re-entrant interpreters are of particular interest when using Push programs as controllers in time sensitive applications. In these situations, Push programs cannot be allowed to run until they are complete or until a loop terminates—there may be strict limits on the number of Push instructions that can be executed per time-step. The re-entrant interpreter allows for the controlled execution of a particular number of instructions per time-step.

3.4 Naming simplified

Previous incarnations of Push allowed names to be bound to values using a `SET` instruction and retrieved later using a `GET` instruction. This allowed, in principle, for evolution of named constants and subroutines but it required synchronization of several different instructions. The introduction of the `EXEC` stack presents opportunities for simplification.

Binding a name to a subroutine has been simplified by one instruction, using the `EXEC` stack instead of a quoted value on the `CODE` stack:

```
Push2:
( TIMES2 CODE.QUOTE ( 2 INTEGER.* )  CODE.SET )

Push3:
( TIMES2 EXEC.DEFINE ( 2 INTEGER.* ) )
```

Executing a subroutine has been simplified by two instructions. The bound symbol is now executed directly (the binding is copied to the EXEC stack), instead of being loaded onto the CODE stack with CODE.GET and executed with CODE.DO:

```
Push2:
( TIMES2 CODE.GET CODE.DO )

Push3:
( TIMES2 )
```

Push3's scheme is considerably more parsimonious. Although none of the examples in this paper make non-trivial use of names, these improvements presumably increase the chances that systems that evolve Push code will be able to make use of named variables and subroutines.

4. EXAMPLES

The following examples were produced using PushGP, a genetic programming system that is generic aside from its representation of evolving programs in the Push programming language. Versions of PushGP implemented in Lisp and C++ are freely available online,[4] as is a version of the Breve simulation environment that includes an embedded PushGP system (based on the C++ implementation).[5] PushGP is a generation-based system that uses tournament selection and nearly-standard genetic operators.[6] Evolution across multiple processors is supported through asynchronous migration of selected individuals between "demes" that run independently, one per node of a computer cluster.

In the runs that produced the results presented below a variety of control parameters were used, with populations ranging from 5,000 to 230,000 (distributed over up to 23 CPUs), tournament sizes ranging from 5 to 7, mutation and crossover rates each ranging from 40% to 45% (with the remainder of each generation produced by straight reproduction and/or immigration), and numbers of generations ranging from 200 to 350. The possibility of unbounded recursion or iteration requires the imposition of execution-step limits (set between 150 and 1000) and program size limits (typically between 100 and 250). Instructions that would violate the program size limit act as no-ops. When a program exceeds the execution-step limit a fitness penalty may be imposed; in some of our runs we imposed a severe penalty (ensuring that no violating program would ever produce offspring) while in others we imposed a mild penalty (allowing violating programs to reproduce, but preferring non-violating programs) or no penalty at all (in which cases a non-terminating program could count as a solution—we note any such cases explicitly below). We used large, general-purpose Push instruction sets, usually excluding only the RAND instructions (which produce random numbers, random code fragments, etc.), some of the higher level code-manipulation instructions (such as CODE.SUBST), and instructions associated with the FLOAT data type (since none of the examples involved floating-point numbers).

[4] http://hampshire.edu/lspector/push.html

[5] http://www.spiderland.org/breve

[6] The operators differ slightly from those of standard genetic programming because Push's syntax involves no distinction between function and argument positions. Some implementations of PushGP also provide "size fair" genetic operators [4].

For many of the problems discussed here PushGP produced large numbers of solutions that used a variety of algorithms based on different Push instructions. In many cases it was possible to coerce the system to produce different styles of solutions by making particular instructions available or unavailable. In particular, most of the problems could be solved without any of the explicit iteration instructions (EXEC or CODE versions of DO*TIMES, DO*COUNT, and DO*RANGE), but most of those solutions were convoluted and did not generalize beyond the inputs used for fitness evaluation.

We simplify the programs produced by PushGP using a simple hill-climbing algorithm that repeatedly performs a random simplification (e.g. the removal of an instruction or expression) and retains the simpler program if it is equally good.

4.1 Reversing a list

For this problem we provide a list of integers, of length between 10 and 30, as input on the CODE stack. A correct program is one that leaves a list with the same elements but the opposite order on top of the CODE stack. We seek a program that correctly reverses any input list, of any length. A similar problem was studied by Olsson in the ADATE system [16]. We used a fitness test with 10 random inputs and based fitness values on the number of elements in the proper positions in the final list.

The following is an evolved, 100% correct, general solution:

```
(CODE.DO*TIMES (CODE.DO* CODE.LIST
(((INTEGER.STACKDEPTH EXEC.DO*TIMES)
(BOOLEAN.YANKDUP CODE.FROMINTEGER))
CODE.FROMINTEGER INTEGER.SWAP)
(CODE.YANKDUP INTEGER.% (BOOLEAN.AND)
CODE.STACKDEPTH EXEC.DO*TIMES)) (CODE.CONS)
(BOOLEAN.SHOVE (CODE.EXTRACT EXEC.S
(EXEC.FLUSH CODE.IF BOOLEAN.YANK
(CODE.FROMINTEGER CODE.ATOM (CODE.SWAP
BOOLEAN.SHOVE (INTEGER.MAX) (CODE.QUOTE
CODE.APPEND CODE.IF)) ((CODE.ATOM CODE.SHOVE
EXEC.POP (CODE.DO*TIMES BOOLEAN.SHOVE) INTEGER.ROT)
(INTEGER.> BOOLEAN.AND CODE.DO* INTEGER.ROT)
CODE.CONS INTEGER.ROT ((CODE.NTHCDR) INTEGER.ROT
BOOLEAN.DUP) INTEGER.SHOVE (CODE.FROMNAME
(CODE.CONS CODE.FROMINTEGER)))) CODE.LENGTH
INTEGER.MAX EXEC.Y)) (BOOLEAN.= (CODE.QUOTE
INTEGER.SWAP) CODE.POP) INTEGER.FLUSH))
```

This solution can be simplified to the following:

```
(CODE.DO* INTEGER.STACKDEPTH EXEC.DO*TIMES
CODE.FROMINTEGER CODE.STACKDEPTH EXEC.DO*TIMES
CODE.CONS)
```

In this program the CODE.DO* instruction "executes" the input list which has the effect of placing all of its elements onto the INTEGER stack. Then the sequence "EXEC.DO*TIMES CODE.FROMINTEGER" moves all of the values onto the code stack and the sequence "CODE.STACKDEPTH EXEC.DO*TIMES CODE.CONS" creates a list of all of the elements. Three loops are used in this solution. The first is really the "execution" of the input list, which is bounded by the length of the input. The second and third are implemented with EXEC.DO*TIMES,

which takes its bound in both cases from the depth of a stack (first the INTEGER stack and later the CODE stack).

4.2 Factorial

In this problem we provide an input integer on the INTEGER stack and we seek a program that leaves the factorial of the input on top of the INTEGER stack. Fitness values were the sum of the errors over all cases (with lower being better). We assessed programs on inputs from 0 to 8, but sought programs that gave correct answers for all possible inputs.

In one of our runs we found a solution that simplified to the following concise, 100% correct, and general program:

```
(1 EXEC.DO*RANGE INTEGER.*)
```

This runs a loop with a counter running from the input down to 1, with each execution of the loop pushing the counter (this is part of the definition of EXEC.DO*RANGE) and then executing the body of the loop, which is simply INTEGER.*. The fact that EXEC.DO*RANGE is so well suited to the computing of factorials is a happy coincidence, but the system also found alternative expressions. For example, it found another 100% correct and general solution that simplified to the following:

```
(INTEGER.* INTEGER.STACKDEPTH CODE.DO*RANGE
INTEGER.MAX)
```

This program relies on the fact that Push programs are pushed onto the CODE stack prior to execution, which gives a program access to its own code for the sake of manipulation, recursive execution, and the like. In this case the program uses the CODE.DO*RANGE instruction to call *itself* the appropriate number of times, executing an INTEGER.* each time and using an INTEGER.MAX each time to pull the product out from under the result of INTEGER.STACKDEPTH, which is used the first time through to produce the 1 which forms the lower bound on the loop counter. The calls to CODE.DO*RANGE within the recursive calls have no effect because the CODE stack is empty at that point.

4.3 Fibonacci

In this problem, versions of which have been explored with several other approaches [9, 14, 12, 13, 37], we provide an input integer on the INTEGER stack and we seek a program that leaves the corresponding element of the Fibonacci sequence on top of the INTEGER stack. We assessed fitness on inputs ranging from 1 to 12, with the following input/output pairs: (1 1) (2 1) (3 2) (4 3) (5 5) (6 8) (7 13) (8 21) (9 34) (10 55) (11 89) (12 144).

Fitness values were the sum of the errors over all cases (with lower being better) and we sought programs that gave correct answers for all possible inputs.

One evolved (100% correct, general) solution simplified to the following:

```
(EXEC.DO*TIMES (CODE.LENGTH EXEC.S)
INTEGER.STACKDEPTH CODE.YANKDUP)
```

This solution uses an EXEC.DO*TIMES loop in conjunction with the EXEC.S combinator to build an expression on the EXEC stack that contains the instruction INTEGER.STACKDEPTH exactly $Fibonacci(N)$ times. When the expression is then executed, each time the INTEGER.STACKDEPTH instruction is encountered it looks at the depth of the INTEGER stack and pushes that value onto the INTEGER stack, thus implementing a simple counter mechanism.

Another evolved (100% correct, general) program exploited the same trick of using the EXEC.S combinator to generate $Fibonacci(N)$ instances of something, in this case copies of the NAME.= instruction on the CODE stack. It then uses CODE.STACKDEPTH to produce the appropriate output. This program simplified to the following:

```
(EXEC.DO*COUNT EXEC.S CODE.QUOTE NAME.=
CODE.DO*COUNT CODE.YANKDUP CODE.DO*COUNT
CODE.CONS CODE.STACKDEPTH)
```

4.4 Parity

In this problem we seek solutions to the *general* even parity problem: given a list of any number of Boolean values on the CODE stack, we seek a BOOLEAN answer of TRUE if an even number of the provided values are TRUE, and an answer of FALSE otherwise.

We used a random selection of 64 inputs of length 8. In one run that included no penalty for non-termination we evolved a solution that simplified to the following:

```
(CODE.DO* EXEC.Y BOOLEAN.=)
```

This program does not terminate until it hits the execution step limit. It solves the even parity problem for even length inputs, and the odd parity problem for odd length inputs. CODE.DO* places all of the inputs onto the BOOLEAN stack. EXEC.Y then triggers an infinite loop of BOOLEAN.=, which hits the evaluation limit but leaves the proper output on the top of the stack.

Because we sought solutions that generalize for all input lengths we conducted additional runs in which the inputs were randomly padded with up to 4 additional FALSEs. We also imposed mild penalties for non-termination.

One evolved (100% correct, general, terminating) solution simplified to the following:

```
((((CODE.POP CODE.DO BOOLEAN.STACKDEPTH)
(EXEC.DO*TIMES) (BOOLEAN.= BOOLEAN.NOT))))
```

In this program CODE.POP removes the program itself from the CODE stack, onto which it was pushed prior to execution. CODE.DO then takes the input values from the CODE stack and, by executing them, places them on the BOOLEAN stack. BOOLEAN.STACKDEPTH then puts the number of inputs on the INTEGER stack, which provides the bound for the EXEC.DO*TIMES loop that uses BOOLEAN.= and BOOLEAN.NOT to produce the correct answer.

4.5 Expt(2, n)

Push3's built-in iteration instructions allow for concise solutions to several of our test problems, but we were also interested in seeing how some of the other control structure primitives might be used to solve essentially iterative problems. We therefore conducted runs on the problem of determining 2^n from an input of n without the use of either the EXEC or CODE versions of DO*TIMES, DO*COUNT, or DO*RANGE. Although we did not find any solutions that generalized beyond the fitness cases in this particular set of runs, we did find several novel approaches to the computation required for error-free performance on the fitness cases ($n = 1$ to $n = 8$). One such approach is embodied in the following evolved solution:

```
((INTEGER.DUP EXEC.YANKDUP EXEC.FLUSH 2
CODE.LENGTH) 8 (2 8  INTEGER.* INTEGER.DUP)
(EXEC.YANK 8 INTEGER.* ((CODE.IF (EXEC.ROT))
BOOLEAN.DEFINE EXEC.YANK)))
```

This first duplicates the input argument and then uses the EXEC.YANKDUP instruction to push onto the EXEC stack a duplicate of an element deep in the EXEC stack. The code that is "yanked" depends on the input:

- case 1: It will yank and then execute "2" (and then, because of the EXEC.FLUSH, it will halt).
- case 2: It will yank and then execute "CODE.LENGTH," pushing the top-level length of the program (4), and then halt.
- case 3: It will yank and then execute "8" (and then halt).
- case 4: It will yank and then execute "(2 8 INTEGER.* INTEGER.DUP)," pushing 16 (twice), and then halt.
- case 5–8: For all of these cases EXEC.YANKDUP will yank out a copy of the large expression on the bottom of the stack, transforming the EXEC stack to:

```
(EXEC.YANK 8 INTEGER.* ((CODE.IF (EXEC.ROT))
BOOLEAN.DEFINE  EXEC.YANK)) EXEC.FLUSH 2
CODE.LENGTH 8 (2 8 INTEGER.* INTEGER.DUP)
(EXEC.YANK 8 INTEGER.* ((CODE.IF (EXEC.ROT))
BOOLEAN.DEFINE  EXEC.YANK))
```

Cases 5–8 then continue as follows:

- case 5: It will yank the CODE.LENGTH, put it in front of "8 INTEGER.*, and as CODE.LENGTH is 4 it will put 32 on the stack. Later EXEC.ROT will take the EXEC.FLUSH and put it before the EXEC.YANK which would otherwise pop the result 32 from the INTEGER stack
- case 6: Instead of CODE.LENGTH, it will yank 8 and do the same as with case 5 (i.e. multiply by 8).
- case 7: It will yank "(2 8 INTEGER.* INTEGER.DUP)," which produces 16, multiply by 8, and then proceed as before.
- case 8: It will yank out the large expression on the bottom of the stack *again*, the cumulative effect of which (through a chain of events too convoluted to recount in detail) will be to calculate the answer 256 and to finish with EXEC.FLUSH.

Note that cases 5, 6 and 7 use the same code for getting the values out as is used for cases 2, 3 and 4. This program uses several forms of code self-manipulation, indexing, and re-use (cases 2, 3 and 4 are applied to 5, 6 and 7). The program also uses the EXEC instructions in interesting ways: EXEC.YANK and EXEC.YANKDUP are used as gotos or subroutine calls, EXEC.ROT is used to manipulate execution flow, and EXEC.FLUSH (which empties the stack) is used to halt the program and to prevent the result from being deleted.

4.6 Sorting a list

The problem of sorting is an interesting one with a history in the literature (for a recent survey see [1]). In an initial attempt to evolve a sort program we provided inputs, which were lists of consecutive but scrambled integers starting with 0, on the CODE stack and looked for the output on the CODE stack after program execution (as was done for the "reversing a list" problem above). We found the following program, which is completely dependent on our peculiar choice of fitness cases:

```
(CODE.LENGTH CODE.POP EXEC.DO*COUNT
(CODE.FROMINTEGER CODE.APPEND))
```

This program actually ignores the input, aside from its length, and generates a list of consecutive integers of the appropriate length. While clever, this was clearly not what we wanted. We used a more diverse set of fitness cases but found that the problem was difficult in the form in which it was presented. We conjectured that this was because the representation demanded fairly sophisticated list processing; for example one has to ensure that no items are lost and that the list's length is maintained while using list-manipulation instructions that can easily violate these constraints. We had better success with an alternative scheme (see [1]) in which we stored the list to be sorted in an external data structure upon which the following instructions could operate:

- INTEGER.LIST-SWAP: Takes 2 integers indices, which are interpreted modulo the list length (as are all indices below), and swaps the indexed elements.
- INTEGER.LIST-LENGTH: Pushes the length of the input list (which is constant for each fitness case) onto the INTEGER stack.
- INTEGER.LIST-GET: Takes an integer index and pushes the indexed number onto the INTEGER stack.
- INTEGER.LIST-COMPARE: Takes two indices and re-pushes the one that indexes the greater element.

Under this re-formulation sort algorithms emerge quite readily. One of the evolved solutions simplified to the following:

```
(INTEGER.LIST-LENGTH INTEGER.SHOVE
INTEGER.STACKDEPTH CODE.DO*RANGE
INTEGER.YANKDUP INTEGER.DUP EXEC.DO*COUNT
INTEGER.LIST-COMPARE INTEGER.LIST-SWAP)
```

This solution was evolved using fitness cases between 4 and 8 elements long but it is 100% correct and it generalizes to arbitrary lists of arbitrary length. For an input of length n it executes exactly $n \times \frac{(n-1)}{2}$ comparisons, which makes it equivalent to many commonly used sort routines (e.g. bubble sort and linear insertion sort). With the addition of an efficiency component to the fitness function it may be possible to evolve novel $O(n \times \log(n))$ sorting algorithms.

5. CONCLUSIONS

The Push3 EXEC stack supports powerful and parsimonious control regimes through explicit manipulation of the

stack of expressions that are queued for execution. These control regimes include standard iteration, several forms of recursion based on code manipulation, combinators, and named subroutines, and less conventional strategies that are difficult to classify. A straightforward genetic programming system that produces Push3 programs (PushGP) can routinely produce solutions that incorporate a range of these control regimes; examples were provided here for reversing and sorting lists and for computing factorials, Fibonacci numbers, powers of 2, and parity. Application of these techniques to real-world problems is currently in progress.

The examples presented here demonstrate the range and novelty of the results that Push3 can routinely produce, but this paper does not provide specific comparisons to other approaches. Comparisons with respect to certain aspects of performance have been provided elsewhere for earlier versions of Push (e.g., scalability and modularization in [28]); the extension of these studies for Push3 is a topic for future work. The primary contributions of Push3, however, concern the overall range of the results that it can produce and the comparative elegance and generality of the mechanisms by which it can produce them. Comparative assessment of these contributions may be best performed after a larger body of results has been produced.

Implementations of the Push programming language and the PushGP genetic programming system are available freely from the Push project home page.[7]

6. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 0308540 and Grant No. 0216344. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation. We thank the anonymous reviewers for their careful reading and helpful comments.

7. REFERENCES

[1] R. Abbott, J. Guo, and B. Parviz. Guided genetic programming. In *The 2003 International Conference on Machine Learning; Models, Technologies and Applications*. CSREA Press, 2003.

[2] P. J. Angeline and J. B. Pollack. Evolutionary module acquisition. In D. Fogel and W. Atmar, editors, *Proceedings of the Second Annual Conference on Evolutionary Programming*, pages 154–163, La Jolla, CA, USA, 25-26 Feb. 1993.

[3] S. Brave. Evolving recursive programs for tree search. In P. J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 10, pages 203–220. MIT Press, Cambridge, MA, USA, 1996.

[4] R. Crawford-Marks and L. Spector. Size control via size fair genetic operators in the PushGP genetic programming system. In W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 733–739, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[5] H. Curry and R. Feys. *Combinatory Logic*, 1, 1958.

[6] K. E. Kinnear, Jr. Alternatives in automatic function definition: A comparison of performance. In K. E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 6, pages 119–141. MIT Press, 1994.

[7] M. J. Kochenderfer. Evolving hierarchical and recursive teleo-reactive programs through genetic programming. In C. Ryan, T. Soule, M. Keijzer, E. Tsang, R. Poli, and E. Costa, editors, *Genetic Programming, Proceedings of EuroGP'2003*, volume 2610 of *LNCS*, pages 84–94, Essex, 14-16 Apr. 2003. Springer-Verlag.

[8] J. Koza. Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems. Technical Report STAN-CS-90-1314, Dept. of Computer Science, Stanford University, June 1990.

[9] J. R. Koza. Hierarchical genetic algorithms operating on populations of computer programs. In N. S. Sridharan, editor, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence IJCAI-89*, volume 1, pages 768–774. Morgan Kaufmann, 20-25 Aug. 1989.

[10] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[11] J. R. Koza, David Andre, F. H. Bennett III, and M. Keane. *Genetic Programming III: Darwinian Invention and Problem Solving*. Morgan Kaufman, Apr. 1999.

[12] K. S. Leung, K. H. Lee, and S. M. Cheang. Evolving parallel machine programs for a Multi-ALU processor. In D. B. Fogel, M. A. El-Sharkawi, X. Yao, G. Greenwood, H. Iba, P. Marrow, and M. Shackleton, editors, *Proceedings of the 2002 Congress on Evolutionary Computation CEC2002*, pages 1703–1708. IEEE Press, 2002.

[13] K. S. Leung, K. H. Lee, and S. M. Cheang. Parallel programs are more evolvable than sequential programs. In C. Ryan, T. Soule, M. Keijzer, E. Tsang, R. Poli, and E. Costa, editors, *Genetic Programming, Proceedings of EuroGP'2003*, volume 2610 of *LNCS*, pages 108–120, Essex, 14-16 Apr. 2003. Springer-Verlag.

[14] M. Nishiguchi and Y. Fujimoto. Evolutions of recursive programs with multi-niche genetic programming (mnGP). In *Proceedings of the 1998 IEEE World Congress on Computational Intelligence*, pages 247–252, Anchorage, Alaska, USA, 5-9 May 1998. IEEE Press.

[15] P. Nordin and W. Banzhaf. Evolving Turing-complete programs for a register machine with self-modifying code. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 318–325, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.

[16] R. Olsson. Inductive functional programming using incremental program transformation. *Artificial Intelligence*, 74(1):55–81, Mar. 1995.

[7] `http://hampshire.edu/lspector/push.html`

[17] T. Perkis. Stack-based genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 148–153, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.

[18] A. Racine, M. Schoenauer, and P. Dague. A dynamic lattice to evolve hierarchically shared subroutines: DL'GP. In W. Banzhaf, R. Poli, M. Schoenauer, and T. C. Fogarty, editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391 of *LNCS*, pages 220–232, Paris, 14-15 Apr. 1998. Springer-Verlag.

[19] S. C. Roberts, D. Howard, and J. R. Koza. Evolving modules in genetic programming by subtree encapsulation. In J. F. Miller, M. Tomassini, P. L. Lanzi, C. Ryan, A. G. B. Tettamanzi, and W. B. Langdon, editors, *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 160–175, Lake Como, Italy, 18-20 Apr. 2001. Springer-Verlag.

[20] J. Schmidhuber. Optimal ordered problem solver. Technical Report IDSIA-12-02, IDSIA, 31 July 2002.

[21] M. Schönfinkel. Über die bausteine der mathematischen logik. *Mathematische Annalen*, 92:307–316, 1924.

[22] L. Spector. Simultaneous evolution of programs and their control structures. In P. J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 7, pages 137–154. MIT Press, Cambridge, MA, USA, 1996.

[23] L. Spector. Autoconstructive evolution: Push, pushGP, and pushpop. In L. Spector, E. D. Goodman, A. Wu, W. B. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. H. Garzon, and E. Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 137–146, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann.

[24] L. Spector. *Automatic Quantum Computer Programming: A Genetic Programming Approach*, volume 7 of *Genetic Programming*. Kluwer Academic Publishers, Boston/Dordrecht/New York/London, June 2004. in press.

[25] L. Spector, J. Klein, C. Perry, and M. Feinstein. Emergence of collective behavior in evolving populations of flying agents. In E. Cantú-Paz, J. A. Foster, K. Deb, D. Davis, R. Roy, U.-M. O'Reilly, H.-G. Beyer, R. Standish, G. Kendall, S. Wilson, M. Harman, J. Wegener, D. Dasgupta, M. A. Potter, A. C. Schultz, K. Dowsland, N. Jonoska, and J. Miller, editors, *Genetic and Evolutionary Computation – GECCO-2003*, volume 2723 of *LNCS*, pages 61–73, Chicago, 12-16 July 2003. Springer-Verlag.

[26] L. Spector, J. Klein, C. Perry, and M. Feinstein. Emergence of collective behavior in evolving populations of flying agents. *Genetic Programming and Evolvable Machines*, 6(1):111–125, Mar. 2005.

[27] L. Spector, C. Perry, J. Klein, and M. Keijzer. Push 3.0 programming language description. Technical Report HC-CSTR-2004-02, Hampshire College School of Cognitive Science, 2004. `http://hampshire.edu/lspector/push3-description.html`.

[28] L. Spector and A. Robinson. Genetic programming and autoconstructive evolution with the push programming language. *Genetic Programming and Evolvable Machines*, 3(1):7–40, Mar. 2002.

[29] K. Stoffel and L. Spector. High-performance, parallel, stack-based genetic programming. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 224–229, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[30] E. Tchernev. Forth crossover is not a macromutation? In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 381–386, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.

[31] P. A. Whigham and R. I. McKay. Genetic approaches to learning recursive relations. In X. Yao, editor, *Progress in Evolutionary Computation*, volume 956 of *Lecture Notes in Artificial Intelligence*, pages 17–27. Springer-Verlag, 1995.

[32] M. L. Wong. Applying adaptive grammar based genetic programming in evolving recursive programs. In S.-B. Cho, H. X. Nguyen, and Y. Shan, editors, *Proceedings of The First Asian-Pacific Workshop on Genetic Programming*, pages 1–8, Rydges (lakeside) Hotel, Canberra, Australia, 8 Dec. 2003.

[33] M. L. Wong and K. S. Leung. Evolving recursive functions for the even-parity problem using genetic programming. In P. J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 11, pages 221–240. MIT Press, Cambridge, MA, USA, 1996.

[34] M. L. Wong and K. S. Leung. Learning recursive functions from noisy examples using generic genetic programming. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 238–246, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[35] G. T. Yu. *An Analysis of the Impact of Functional Programming Techniques on Genetic Programming*. PhD thesis, University College, London, Gower Street, London, WC1E 6BT, 1999.

[36] T. Yu. Hierachical processing for evolving recursive and modular programs using higher order functions and lambda abstractions. *Genetic Programming and Evolvable Machines*, 2(4):345–380, Dec. 2001.

[37] T. Yu. A higher-order function approach to evolve recursive programs. In *Genetic Programming Theory and Practice 3*. Kluwer, 2005. To appear.

[38] T. Yu and C. Clack. Recursion, lambda abstractions and genetic programming. In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 422–431, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.