Genetic Programming and Evolvable Machines, 3, 7–40, 2002

Genetic Programming and Autoconstructive Evolution with the Push Programming Language

LEE SPECTOR lspector@hampshire.edu
ALAN ROBINSON Alan@HciDesign.com
Cognitive Science, Hampshire College, Amherst, MA 01002

Received August 15, 2001; Revised November 26, 2001

Abstract. Push is a programming language designed for the expression of evolving programs within an evolutionary computation system. This article describes Push and illustrates some of the opportunities that it presents for evolutionary computation. Two evolutionary computation systems, PushGP and Pushpop, are described in detail. PushGP is a genetic programming system that evolves Push programs to solve computational problems. Pushpop, an "autoconstructive evolution" system, also evolves Push programs but does so while simultaneously evolving its own evolutionary mechanisms.

Keywords: genetic programming, artificial life, modularity, programming languages, self-adaptation

1. Introduction

Several forms of genetic and evolutionary computation produce computer programs through evolutionary processes. The Push programming language was designed as a language for the programs that evolve in such systems.

Previous work has evolved programs in many forms including Lisp expressions [18], machine code strings [26], purely functional expressions [48, 50], graph-based constructions [45], object hierarchies [9], fuzzy rule systems [46, 29], logic programs [27], and specialized assembly languages [1, 31], among others. Each of these representations has advantages and disadvantages but the wide scope of this past work, covering nearly every well-known programming paradigm, is impressive. One would be justified in assuming that this ground has been covered and that little is to be gained from *yet another* program representation for evolved programs.

This article will argue, to the contrary, that a new language for evolved programs can make a substantial difference. It can improve existing evolutionary computation methods and it can also enable the development of entirely new methods with novel properties.

In particular we describe a language called Push which, when used in an otherwise ordinary genetic programming system (called PushGP), *automatically* provides advanced genetic programming facilities (multiple data types, automatically defined subroutines, control structures and architecture) without extra machinery or user configuration. We also show how Push supports a new, radically self-adaptive form of evolutionary computation called *autoconstructive evolution* (defined in Section 2.4 below) that we illustrate with the Pushpop system.

In the Section 2 we describe the general goals of the Push project and the specific goals motivating the development of PushGP and Pushpop. Section 3 is a detailed introduction to the Push programming language with several code examples. Section 4 describes PushGP and presents evidence that it automatically evolves programs that capitalize on the modularity of the problems that they were evolved to solve. Section 5 describes Pushpop and presents initial data documenting the self-adaptive evolution of its reproductive mechanisms. A brief language reference for Push is provided following our conclusions.

2. Motivation

2.1. Towards autonomous evolution

Evolutionary computation is still a "black art," the successful application of which requires considerable experience and knowledge. Such reliance on human expertise is antithetical to the general goals of the field; we seek *automatic* techniques that *learn* or *evolve* solutions to problems with minimal human intervention.

Autonomy is clearly important when evolutionary computation is used in an effort to understand the nature of biological evolution; indeed, autonomy may be the single most characteristic feature of the natural systems that we seek to model. But it is also important when evolutionary computation is used as a method for solving practical problems, as we seek an engineering methodology that requires as little as possible from the human user.

The Push project was initially driven not by an interest in programming languages *per se* but by the goal of making evolutionary computation systems more autonomous. This is a goal shared with many other recent projects in the field (usually labeled "self-adaptive"—see, for example, [2, 4, 6, 16, 41]). The particular self-adaptive concept at the core of the Push project, autoconstructive evolution (defined in Section 2.4 below), requires that evolving programs contain the code for their own reproduction and diversification. This, it turns out, puts heavy demands on the language in which the programs are expressed. At very least such a language must support general code manipulation and there are other requirements as well (see Section 3.1 below). The Push language was developed in response to this demand, to support the development of the autoconstructive evolution system called Pushpop.

Conveniently, the features of Push that support autoconstructive evolution can also provide self-adaptive capabilities to more traditional forms of evolutionary computation. When used within a standard genetic programming system Push provides several forms of self-adaptation including the automatic evolution of modules and program architecture. The system that we have built to test these ideas, PushGP, appears to have promise as a problem-solving technology independent of its connection to autoconstructive evolution.

2.2. Evolution of multi-type programs

Complex programs usually manipulate data of more than one type. If one were to survey garden-variety "real world" programs one would probably find that the overwhelming majority manipulate integers, floating point numbers, Boolean values, and strings of characters. A smaller majority undoubtedly use many more types, including hierarchically defined types such as "arrays of arrays of integers."

With the availability of multiple types comes the danger that operations can be applied to inappropriate data. In a world containing only integers, for example, one could confidently apply + to anything, but in a world containing, say, hierarchical lists of symbols one must be more careful; addition (of the normal sort) simply isn't applicable to such data. In most popular languages syntax restrictions help to prevent operations on inappropriate data. In some languages and in some circumstances runtime type checks catch type errors that survive the syntax checks, and modern languages also provide "exception handling" facilities that allow for graceful recovery from these and other errors.

Because of these complications the first programs evolved by genetic programming manipulated data of only one type. In genetic programming one "slices and dices" code in random ways during mutation and crossover, and it is important that the scrambled code can be counted on to parse correctly, to terminate properly, and to produce potentially meaningful results. This is most easily achieved by adopting a uniform syntax (Lisp symbolic expressions in standard genetic programming) and by restricting all values (including the inputs and possible outputs of all functions that can appear in evolved programs) to the same type.

Montana has developed a popular strategy for incorporating multiple data types in evolving programs called *strongly typed genetic programming* [25]. In strongly typed genetic programming the user is required to specify the types of all values, function inputs, and function outputs, and the program generation, mutation, and crossover algorithms are modified to obey these type restrictions. Strongly typed genetic programming has been used in a wide range of practical applications. It complicates the development of genetic operators but this is normally a modest cost considering the benefits gained. It also certainly affects the shape of the program search space (for example by restricting crossover points) and it presumably thereby affects evolutionary dynamics, but it is not yet clear under what circumstances this may be beneficial or detrimental. If one is attempting to *evolve* genetic operators, however, then it is clear that syntax restrictions, including those required by strongly typed genetic programming, are problematic. In Push multiple data types are accommodated without syntax restrictions or the attendant impacts on code generation, genetic operators, or the program search space, and the evolution of genetic operators is also greatly simplified.

2.3. Evolution of modularity, recursion, and control structure

Complex programs are usually modular. In most programs many operations are performed many times, executing identical code each time (though possibly with different parameters). It therefore usually makes sense to collect repeated code into subroutines that can be defined once and then used many times. This kind of modularization is widely recognized as a central pillar of good programming practice; it contributes not only to more concise code but also to programs that are easier to write, understand, debug, and extend. Many advances in software engineering (for example object-oriented design and programming) are essentially applications of the general concepts of modularity.

As with the mechanisms for multiple data types described above, most mechanisms for defining and using modules involve syntax restrictions; for this reason early genetic programming systems evolved only un-modularized programs. Several researchers later showed that the ability to evolve modules greatly extends the power of genetic programming, and several different schemes for evolving modules were developed including the *genetic library builder* [5], *automatically defined functions* [19], *automatically defined macros* [37], and *adaptive representation through learning* [33]. All of these schemes required modifications to the genetic programming algorithm and most required pre-specification of the architecture (number of modules, number of inputs to each module, etc.) of the evolved programs. Later work by Koza et al. [20] on *architecture-altering operations* eliminated the need for architecture pre-specification for automatically defined functions but did so at the cost of additional complexity of the genetic programming algorithm.

Push's uniform syntax, in conjunction with its facilities for handling multiple data types, provides a new and elegant solution to these problems. In particular, a `CODE` data type, handled similarly to all other types, allows Push programs to define and to execute modules as they run, without imposing syntax restrictions. Modularity comes "for free" with the `CODE` type and any module architecture can be expressed. Recursion also comes "for free" but of course the possibility of non-terminating recursive code requires that we limit runtime [8, 50].[1]

2.4. *Evolution of evolutionary mechanisms*

In genetic algorithms and genetic programming the methods of reproduction and diversification are designed by human programmers; the reproductive systems do not themselves evolve. High-performing individuals are reproduced, mutated, and recombined using pre-specified algorithms with pre-specified parameters (like mutation rates). These algorithms and parameters may not be optimal, 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, the evolution of life on Earth presumably began without the imposition of pre-specified mechanisms for reproduction and diversification. Reproductive systems co-evolved with the mechanisms underlying all of the other functions of the earliest organisms, including those for finding and making use of 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 [23].

We define an *autoconstructive evolution system* to be any evolutionary computation system that adaptively constructs its own mechanisms of reproduction and diversification as it runs. Note that such systems thereby evolve their own evolutionary mechanisms, much as early life on Earth must have evolved its own evolutionary mechanisms.

There are at least two motivations for exploring the development of autoconstructive evolution systems. The first is that such systems might evolve evolutionary mechanisms that are better adapted to particular classes of problems than hand-designed mechanisms. We can hypothesize that autoconstructive evolution systems, if designed well, may be capable of out-performing traditional evolutionary computation systems by adapting their reproductive mechanisms to their representations and problem environments. The second motivation is that such systems are unique as models of natural evolution and that they may thereby provide 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" [2, 4, 6, 16, 41]. 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. Edmonds, in his "Meta-Genetic Programming" framework, took a more radical approach 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 [12]. As Edmonds noted, there is a potential regress here, as genetic operators are required for the evolution of his evolved operators. Edmonds proposed several approaches to the regress problem including recursive strategies in which operators act on the populations in which they are evolved. He 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 [45]. More recently, Kantschik and colleagues have extended Teller's ideas and reported success with a meta-genetic programming system operating on graph-based programs [17].

Autoconstructive evolution takes one step closer to the natural model by 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. Problem-solving and child-producing components of 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 including work on the Tierra, Avida, and Amoeba systems. In Tierra, assembly language programs compete for execution time and evolve on a simulated MIMD parallel computer [31]. The evolving "creatures" in Tierra are 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 problem-solving orientation of autoconstructive evolution (and of Avida—see below) obviously improves the prospects for the practical application of the technology, but it also serves a purpose when these systems are used to study principles of biological evolution. This is because progress in solving a difficult computational problem can serve as an indicator of the capacity of a system for adaptive evolution.

The Avida system extends the ideas of Tierra in several ways, one being that it incorporates a mechanism for guiding evolution to solve problems [1]. In published reports this has been applied only to simple logical or arithmetic problems and it requires that the user run the system in stages of increasing task difficulty. Like Tierra, Avida requires hand seeding and relies on externally specified mutation mechanisms. It also 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 [22]. One recent application of Avida tested a prediction of quasi-species models that high mutation rates can favor the "survival of the flattest"—that is, that low-performance individuals can out-compete high-performance individuals if they are more robust to mutations [49].

The Amoeba system is similar both to Tierra and to Avida but it uses a simplified instruction set. This allows for the spontaneous generation of replicators without hand seeding, but it does so at the cost of computational universality [28].

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" [11, p. 204]. The Dittrich and Banzhaf system exhibits complex evolutionary dynamics without hand seeding 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.[2]

Many studies have been conducted on general principles of self-replication, beginning with von Neumann's work in the 1940's ([35] contains a survey; an accessible introduction is [36]). 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) [18]. Few of the prior studies consider in detail the issues involved in simultaneously evolving self-replication and additional problem-solving functionality.

As mentioned above, the particular approach to autoconstructive evolution described in this work relies upon the use of a programming language for the evolving programs with certain unusual properties. The next section introduces Push, the programming language developed to fill this need, 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. We then return to autoconstructive evolution and describe Pushpop, an autoconstructive population of Push programs.

3. The Push programming language

This section describes the Push programming language and presents some examples of Push programs. The Appendix at the end of the article is a language reference that provides more details, including descriptions of all Push instructions.

3.1. General design goals

The motivations laid out in the previous section lead to several design goals for our programming language. In particular we want a language that is:

expressive We seek a language in which it is easy to represent programs that include:

- multiple data types,
- modules,
- complex control structures including recursion.

self-manipulating For autoconstructive evolution it is necessary, by definition, that programs manipulate and produce other programs. Facilities for code manipulation can also help to provide some of the "expressive" features listed above.

syntactically uniform Uniformity is particularly helpful in easing the development of self-manipulating code, as described above, but it also simplifies other operations in genetic programming.

The Push language meets these design goals by extending concepts of stack-based programming languages that have been used in prior genetic programming research.

3.2. Stack-based programming

Push is a stack-based programming language that might be viewed as a descendant of Forth [34]. Other stack-based languages, including Forth, have been used in previous genetic programming systems; see [10] for a survey.

The defining feature of a stack-based language is that arguments are passed to instructions via global data stacks. This contrasts with argument passing techniques based on registers or activation records. Stack-based argument passing leads naturally to a postfix syntax in which the programmer first specifies (or computes) arguments that are pushed onto the stack and then executes an instruction that uses those arguments. For example, the following is a postfix representation of the addition of 2 and 3:

```
2 3 +
```

This code specifies that 2 and then 3 should be pushed onto the stack and that the + instruction should then be executed. The + instruction removes the top two elements of the stack, adds them together, and pushes the result (5) back onto the stack.

If extra arguments are present they are ignored automatically—each instruction takes only the arguments that it needs from the top of the stack. For example the following will also leave 5 on top of the stack but in this case the 5 will be on top of a 1 that is ignored by +:

```
1 2 3 +
```

Longer calculations can often be expressed in a variety of ways. For example each of the following lines sums the integers from 1 to 4:

```
1 2 3 4 + + +
1 2 + 3 + 4 +
1 2 + 3 4 + +
1 2 3 + 4 + +
```

What if the stack contains too few arguments for an instruction? Under normal circumstances, with a human programmer, a reasonable response would be to signal a run-time error and to transfer control to a debugger. In such circumstances a stack underflow is probably a programming mistake that should be corrected. In an evolutionary computation system, however, this need not be the case.

One approach to stack underflow in an evolutionary computation context would be to ensure that programs with underflows are never executed in the first place. This can be done by restricting code generation and code transformation operators like mutation and crossover [44].

In keeping with the design goal of syntactic uniformity described above, Push instead follows a "permissive execution" precedent established in previous stack-based genetic programming work (e.g. [30, 42]). In Push an instruction with insufficient arguments is simply ignored (treated as a `NOOP`). For example the program "`3 +`" would leave 3 on top of the stack; the `+` instruction would have no effect because it requires two arguments and only one is present.

3.3. Multiple data types

Push handles multiple data types by providing a stack for each type. There is a stack for integers, a stack for floating point numbers, a stack for Boolean values, and so forth. Each instruction takes the inputs that it requires from whichever stack or stacks are appropriate and pushes outputs onto whichever stack or stacks are appropriate. The Boolean `AND` instruction, for example, takes two Boolean values from the `BOOLEAN` stack and pushes one Boolean value (`TRUE` if both of the popped arguments were `TRUE` and `FALSE` otherwise) back onto the same stack. The integer `+` instruction, on the other hand, takes its inputs from and pushes its output onto the `INTEGER` stack. The integer comparison function $<$ takes its inputs from the `INTEGER` stack but pushes its output (`TRUE` if the second popped integer is less than the first popped integer and `FALSE` otherwise) onto the `BOOLEAN` stack.

Literals appearing in code (for example "`TRUE`" or "`3.14`") are pushed onto the appropriate stacks; a simple set of parsing rules serves to recognize the appropriate type for each literal.

Push, like many modern programming languages, organizes its types hierarchically and allows sub-types to inherit functionality from their super-types. Push is not significantly "object-oriented" aside from this hierarchical type system and occasional uses of object-oriented terminology; for example we call different type-specific versions of an instruction the *methods* of the instruction.

Push disambiguates different methods of an instruction by consulting the `TYPE` stack. `TYPE` in Push is a type like any other and as such it has its own stack. Type literals (the names of the types, for example `INTEGER`, `BOOLEAN`, and `TYPE`) are defined for all Push types and these are the items that can be pushed on the `TYPE` stack. When an instruction with multiple methods is executed the `TYPE` stack is consulted (but not popped) to determine which method to execute. For example, consider the following code:

```
1 2 3.0 4.0 INTEGER + FLOAT *
```

Push includes both integer and floating point versions of both the `+` (addition) and $*$ (multiplication) operators. When the interpreter sees the `INTEGER` type literal it pushes it onto the `TYPE` stack. When it then sees the `+` instruction it consults the `TYPE` stack, notes that the top item is `INTEGER`, and executes the integer method for `+`. By the time it gets to $*$ the type on top of the `TYPE` stack will be `FLOAT`, so a floating point multiplication is performed. The result is that after executing this line of code the top item on the `INTEGER` stack will be 3 and the top item on the `FLOAT` stack will be 12.0.

Several generic stack-manipulation instructions, like `POP` and `DUP` (duplicate the top element) are defined for `PUSH-BASE-TYPE`, the root of the type hierarchy, and are therefore available for use on all types.

Because the system consults but does not pop the `TYPE` stack when deciding which method to execute, a single type literal suffices to specialize an arbitrarily long string of instructions. Of course the `TYPE` stack can be popped explicitly if this is desired, using the code "`TYPE REP POP`".

Sometimes the type on top of the `TYPE` stack will not be appropriate for the current instruction; for example in the code "`INTEGER 1 2 BOOLEAN +`" `BOOLEAN` will be on top of the `TYPE` stack when the `+` instruction is executed. In such circumstances the interpreter looks further down in the `TYPE` stack until it finds a type for which the current instruction has a defined method.

To ensure that a method can always be chosen for any instruction a predefined "type stack bottom" is appended to the TYPE stack whenever it is consulted. If the running program did not push appropriate type literals onto the TYPE stack, or if it popped all of the appropriate type literals off of the TYPE stack before getting to the current instruction, then the type stack bottom will provide the necessary default types.

A provided CONVERT instruction is implemented for all *pairs* of types—it operates on the top two types on the TYPE stack. For many pairs of types the conversion rules are obvious (e.g. floating point numbers are truncated for conversion to integers) but in other cases they are necessarily ad hoc. See the Appendix for details.

The types that have been implemented in Push to date are simply those that have been needed for the problems on which we have worked; this is not a "complete" set of types in any sense and we intend to add new types as the need arises. The language's implementation (and in particular the hierarchical type system) makes such extensions relatively easy. Push does not currently include a mechanism whereby new, derived or composite types can be constructed on the fly within Push code (analogously to Lisp's DEFTYPE or similar mechanisms in many other languages) but one can imagine how such a construct might be added to Push and it would be fully consistent with Push's design goals to do so. Such a mechanism would conceivably allow for the evolution of new data structures, as Langdon has done with more conventional techniques [21], which attain the status of complete data types (with their own stacks and instructions). Of course even without such a mechanism one can leverage certain types to implement other data structures; for example, one could use the currently implemented EXPRESSION type (see below) to implement general lists, stacks, queues, etc.

3.4. Code structure

Parentheses in Push have no direct effect on execution, so for example `(+ 2 3)`, `(+ (2 3))`, and `((+) 2 ((3)))` all produce identical results. Parentheses can have indirect effects, however, as they serve to group and provide hierarchical structure to code. This can be of great utility when code is manipulated as data (and possibly later executed—see below).

Comments can be inserted in Push code using the Common Lisp syntax [15]: a semicolon (";") indicates that the remainder of the line is a comment, and block comments are started with "#|" and ended with "|#".

3.5. The code type

Many of Push's most sophisticated features stem from the treatment of code as data that can be manipulated and possibly executed. CODE is a type like any other in Push, and it therefore has its own stack. CODE inherits functionality from the more general EXPRESSION type. A rich set of list-manipulation instructions, inspired by those that form the core of the Lisp programming language, 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 (leaving the CODE argument on the stack until the recursive execution is complete; DO* is the same except that it pops the CODE argument first).[3] 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 program which recursively computes the factorial of an input provided on the INTEGER stack:

```
(QUOTE (POP 1)
 QUOTE (DUP 1 - DO *)
 DUP 2 < IF)
```

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 < instruction, which will have compared the input to 2. When the input is less than 2 it is popped and replaced with 1, the proper answer for the base case of the recursion. When the input is greater than or equal to 2 the interpreter will execute the expression "(DUP 1 - DO*)" and it will do so in an environment in which the *entire program* is once again on top of the CODE stack. The DO instruction will therefore execute the entire program again, but with a decremented input, and the * instruction will multiply the input by the result of the recursive call.

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 [3, 39, 40], 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.

3.6. The name type

The 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 and control structures). Any symbol that is not an instruction name or literal of any other type (including type literals) is considered a NAME literal and is pushed on the NAME stack for use with subsequent calls to SET or GET. For example, the following code stores 3.14 in a variable named PI which is then used to compute the area of a circle with radius provided on the FLOAT stack:

```
FLOAT          ;; specialize following code for floating point
3.14 PI SET    ;; set the PI variable to 3.14
DUP * PI GET * ;; compute PI * R^2
```

In this case the use of a named variable provides no real economy, but in cases when a value must be used many times or when stack-based storage is awkward the mechanism for named variables can be of considerable value.

A single name can be bound to different values for different types; the interpreter maintains an independent name/binding space for each type. Names can be used in conjunction with the CODE type to provide named subroutines or control structures. For example, the following code defines a named factorial subroutine:

```
CODE QUOTE (QUOTE (POP 1)
            QUOTE (INTEGER DUP 1 -
                   CODE FACTORIAL GET DO
                   *)
            INTEGER DUP 2 < IF)
FACTORIAL SET
```

After executing this definition the FACTORIAL subroutine can be called as follows:

```
5                      ;; input
CODE FACTORIAL GET DO  ;; call
```

This call will leave the factorial of 5 (that is, 120) on top of the INTEGER stack.

This module-definition technique can be used not only to define ordinary subroutines but also to define new control structures that would ordinarily require “macro” definition facilities [37]. For example, suppose that a programmer wants a “while” loop structure similar to those provided in many other programming languages. In Push the natural idiom for applying such a control structure would be to push a piece of code representing the loop body onto the CODE stack, followed by a piece of code representing the loop condition. One would then retrieve and execute the definition of WHILE. The following definition of WHILE would make this possible:

```
CODE
QUOTE (CODE CONDITION SET BODY SET ;; bind arguments
       CODE BODY GET CONDITION GET ;; re-push arguments for
                                        recursion
       BODY GET WHILE GET APPEND   ;; push body and recursive
                                        call
       QUOTE (CODE POP POP)        ;; push cleanup code for
                                        termination
       CONDITION GET DO            ;; execute condition
      IF)                          ;; branch appropriately
WHILE SET
```

After this definition one could use the WHILE structure as follows:

```
1                          ; input
QUOTE (INTEGER 2 *)        ; loop body
QUOTE (INTEGER DUP 50 < )  ; loop condition
CODE WHILE GET DO*         ; execute while loop
```

This example starts with 1 on the INTEGER stack and then repeatedly doubles the number on top of the INTEGER stack as long as it remains less than 50. When the loop terminates the number on top of the INTEGER stack will be 64.

The definition of WHILE provided above is not perfect. As in most languages, one must be careful to avoid "variable capture" in macro definitions [14], and the definition above does not do this. In fact the situation here is particularly bad since Push's name bindings are all global. This means that calls to WHILE cannot be nested if WHILE is defined as above. One can fix this either by ensuring that unique variable names are used for each recursive call (using ``NAME RAND'') or by using the stack, rather than named variables, to store all of the arguments. Push allows for both of these solutions but the code for a more robust WHILE structure will probably be more complex.

4. Genetic programming with PushGP

4.1. PushGP

The Push programming language may be useful in a range of evolutionary computation systems. Later in this article we show how it can be used in an autoconstructive evolution system in which evolving programs are responsible for the production of their own offspring. In the present section we show how it can be used in a more conventional way, as the target language for an otherwise-ordinary genetic programming system called PushGP.

The basic algorithm of PushGP is the same as that for standard genetic programming [18]. PushGP begins by generating a population of random programs, using the algorithm listed in Table 1.[4] Each of the randomly generated programs is then evaluated for fitness with respect to the target problem. If a sufficiently fit program (a solution to the target problem) is found then 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. The process continues until a solution is found or until the maximum number of generations has been exceeded.

PushGP uses tournament selection [7] and reasonably standard genetic operators although the flexibility of the Push syntax invites the development of novel operators as well. In most of our experiments we use a traditional mutation operator that replaces a randomly chosen sub-expression with a new random subexpression. We also use a traditional crossover operator that replaces a randomly chosen sub-expression with a randomly chosen sub-expression from another individual. Nodes are chosen for both of these operators using uniform random selection. Alternative operators that we have explored include mutation operators that only replace atoms (instructions or literals) with other atoms and crossover operators that maintain the structure of one parent while performing uniform crossover on atoms that make up the parents.[5] Some of our experiments with alternative operators are documented in [32].

Table 1. Pseudocode for the Push random code generator

```
Function RANDOM-CODE (input: MAX-POINTS)
  Set ACTUAL-POINTS to a number between 1 and MAX-POINTS,
    chosen randomly with a uniform distribution.
  Return the result of RANDOM-CODE-WITH-SIZE called with input
    ACTUAL-POINTS.
End

Function RANDOM-CODE-WITH-SIZE (input: POINTS)
  If POINTS is 1 then choose a random element of the instruction
     set. If this is an ephemeral random constant then return a
     randomly-chosen value of the appropriate type; otherwise
     return the chosen element.
  Otherwise set SIZES-THIS-LEVEL to the result of DECOMPOSE
      called with both inputs (POINTS - 1). Return a list
      containing the results, in random order, of
      RANDOM-CODE-WITH-SIZE called with all inputs in
      SIZES-THIS-LEVEL.
End

Function DECOMPOSE (inputs: NUMBER, MAX-PARTS)
  If NUMBER is 1 or MAX-PARTS is 1 then return a list
      containing NUMBER
  Otherwise set THIS-PART to be a random number between 1 and
      (NUMBER - 1). Return a list containing THIS-PART and
      all of the items in the result of DECOMPOSE with inputs
      (NUMBER - THIS-PART) and (MAX-PARTS - 1)
End
```

4.2. Illustrative examples

In this section we briefly present two examples of PushGP solving simple problems. These examples were chosen to highlight the unique features of the Push language and to demonstrate the ability of PushGP to evolve programs that make use of these features.[6]

4.2.1. 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. 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.[7]

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.[8] 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.[9]
- In this case the appropriate expression stack is the `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.

4.2.2. 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. 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 [19]. 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. This was done as a first step toward conformance with the function set used in [19], although other elements in our instruction set (e.g., those for code manipulation) make this conformance less than complete. See Section 4.3 below for a more careful comparison of PushGP and traditional genetic programming systems on parity problems.

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.[10] 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 literal 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.

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 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.

4.3. Comparison to standard genetic programming with respect to modularity and scaling on parity problems

Koza [19] selected even parity problems as a benchmark for genetic programming with and without automatically defined functions (ADFs) for several reasons:

- Parity problems are easily scaled in difficulty just by changing the arity (the size of N). Furthermore, as the problem scales in difficulty the fundamental nature of the problem remains the same. There is less reason, therefore, to think that differences in the performance at higher arity parity are the results of odd changes in the representation or dynamics at that particular arity.
- Parity problems are difficult for machine learning systems in general, not just genetic programming.
- Parity problems can be solved via modularization, and the modularization of a parity problem makes it much easier to solve from a human standpoint. Typically, parity solutions of high arity are calculated with the aid of lower order parity functions. If ADFs provide an effective method for evolving modularity, then their computational advantage should be measurable on the parity problem.

It is the last point that makes parity particularly interesting as a test of the modularity of PushGP. If PushGP can evolve modularity, and thereby scale well in solving parity problems, then it might also be expected to scale well on other, more complex problems. It is for the sake of this comparison to standard genetic programming with respect to modularity and scaling that we investigate the parity problem and, in particular, Koza's parity results; the use of genetic programming for parity problems has been studied extensively by others since Koza's pioneering work (see for example [50]) but it matters little for our present purposes that Koza's work on this topic is no longer state-of-the-art.

4.3.1. Koza's work with even-parity. Koza selected the even-parity problem to demonstrate the potential positive effects of allowing genetic programming to evolve modular programs. This was done by solving even-parity problems of increasing arities, both with and without ADFs, and comparing the amount of computational effort required in the two conditions.[11] In order to allow the most direct comparison between conditions all GP parameters were kept constant except for the fitness cases and the presence of ADFs.

Since the calculation of computational effort requires that at least some runs complete successfully, it was necessary to use parameters which allowed GP without ADFs to solve the most difficult parity problem; therefore, a large population size of 16,000 was used. This number was more than necessary for GP with ADFs, but was not, in fact, large enough to allow GP without ADFs to solve even-six-parity within the amount of machine time available.

The terminals used for this problem were each of the Boolean inputs for the current arity, and the functions were `AND, OR, NAND`, and `NOR`. The fitness cases consisted of each possible combination of Boolean true and Boolean false for the arity of the problem (i.e., 16 combinations for even-4-parity, and 32 for even-5-parity, etc.). Fitness was the sum of the number of incorrectly classified fitness cases. The success criterion for this problem was 0 fitness. A variety of statistics were collected on several runs of genetic programming on this problem, including the computational effort measure that Koza defined.

Koza obtained the computational effort results shown in Table 2. Note that for Arity 6, no successful programs were found without ADFs. The results are based upon programs that came closest to solving the problem—any successful run would almost certainly require more computational effort. Koza's results, including additional data not reproduced here, show that adding ADFs (that is, allowing GP to evolve modular solutions) had three distinct benefits:

- Less computational effort, by orders of magnitude, as the problems become harder.
- Fewer generations of evolution required to find solutions.
- Smaller programs.

4.3.2. Evaluating the modularity produced by pushGP on even-parity problems. PushGP was used to solve even-parity problems using similar settings as those used by Koza. The goal was to determine if PushGP exhibits the same sorts of performance gains as GP with ADFs via the generation of modular code.

PushGP can potentially create modular code via a number of mechanisms. Because the removal of those mechanisms would severely limit the functionality of the language (and in particular the ability to create and execute modules), we used a large and rich instruction set for all of our runs. The assessment, then, of whether PushGP's ability to create modular code is beneficial must be measured in terms of how the computational effort scales with harder problems. Genetic programming with ADFs drastically reduced the additional amount of effort required to solve higher arity problems relative to genetic programming without ADFs. If PushGP's methods of building modularity are effective then its computational effort should scale at similar rates.

We ran PushGP's on even-parity problems of arity 3, 4, 5, and 6, using the instruction set listed in Table 3. Note that we cannot use an instruction set that is equivalent to Koza's in any significant way while still allowing for the evolution of modularity

Table 2. Computational effort for even-parity problems found by Koza [19]

Arity	Effort without ADFs	Effort with ADFs
3	96,000	64,000
4	384,000	176,000
5	6,528,000	464,000
6	*70,000,000*	1,344,000

Table 3. PushGP instruction set used for even-parity problems

```
Instructions:
DO DO* IF MAP QUOTE ATOM NULL CAR CDR CONS LIST APPEND NTH NTHCDR
MEMBER POSITION SUBST CONTAINS CONTAINER REPLACE-ATOMS INSERT
EXTRACT LENGTH SIZE DISCREPANCY NAND NOR AND OR NOT PULL / + - *
< > DUP POP SWAP REP = NOOP SET GET CONVERT

Type literals:
CHILD CODE NAME TYPE BOOLEAN FLOAT INTEGER

Ephemperal random constant specifiers:
EPHEMERAL-RANDOM-SYMBOL EPHEMERAL-RANDOM-BOOLEAN EPHEMERAL-RANDOM-FLOAT
EPHEMERAL-RANDOM-INTEGER
```

Note. See the Appendix for descriptions of individual instructions. `NAND` and `NOR` were each included twice in the function set, thereby increasing their prevalence in randomly generated code.

in PushGP; this is because modularity in PushGP emerges through the action of a wide range of instructions (including code-manipulation instructions) that actually occur in evolving programs, rather than through the imposition of modular structure by the genetic programming system. For this and other reasons we must be particularly careful in drawing comparisons between Koza's results and the PushGP results presented here. The other parameters of our runs are listed in Table 4. The results of the runs are listed in Table 5.

It is clear that providing the PushGP instruction set in Table 3 for these runs gave PushGP a considerable advantage over the instruction set used by Koza. In all but the easiest problem, PushGP performed better than standard genetic programming with or without ADFs. Indeed, the PushGP instruction set was so well suited to the problems that a large number of solutions were found in the initial random populations (the second column of the table), particularly for arity 3. Again, therefore, we must caution against any simple-minded comparison with standard genetic programming.

Table 5 nonetheless contains data suggesting that PushGP is successfully discovering opportunities for modularity provided by parity problems. The fourth column of the table lists the percentage of successful programs that use the `DO` or `DO*` instructions, which are among the most useful PushGP instructions for modularity. While only one in five solutions to even-3-parity used `DO` or `DO*`, more than half of the solutions to the larger arity problems used these instructions. The apparently

Table 4. Parameters for PushGP runs on even-parity problems

Fitness cases	All possible combinations of the 3, 4, 5, or 6 Boolean inputs
Fitness	The number of incorrectly classified fitness cases
Termination condition	Fitness = 0
Population size	16,000
Maximum generations	100
Maximum program length	100 points
Execution limit	200 points
Genetic operators	45% crossover,45% mutation, 10% exact reproduction
Number of runs	100 for each arity

Table 5. Results of PushGP runs on even-parity problems with the instruction set in Table 3

Arity	Effort	% random solutions	% using `DO` or `DO*`	Effort relative to Koza without ADFs	Effort relative to Koza with ADFs
3	80,000	49	20	1.2X	1.5X
4	96,000	23	62	0.25X	0.55X
5	352,000	3	56	0.54X	0.76X
6	160,000	4	63	0.002X	0.12X

Note. Each row was calculated on the basis of 100 independent runs.

anomalous fact that even-5-parity was more difficult than even-6-parity seems to be due to a better fit between the instruction set and even (as opposed to odd) arities of this problem; multiple equality operators alone solve even arity parity problems, while odd arity parity requires equality operators and also negation.

Perhaps the most useful information is provided in the final two columns of Table 5. Because of the different instruction sets the absolute comparison of computational efforts is not meaningful, but the way in which the effort grows for larger arities is indeed suggestive. Note that PushGP's performance improves more dramatically than standard genetic programming as the arity of the parity problem is increased. The striking improvement in scaling relative to Koza's experiments without ADFs was expected and suggests that PushGP is building modules well-suited to the problem. The improvement relative to Koza's experiments with ADFs is also quite striking (over an order of magnitude) and this was more of a surprise. This improvement may indicate that PushGP is particularly good at discovering opportunities for modularization but we must be careful in drawing such conclusions; another interpretation is that our use of a different and more powerful instruction set simply makes the problem *different* and coincidentally easier to scale.

We explored this alternative interpretation in some detail by studying the evolved programs. We discovered that the PushGP equality instruction is extremely useful in determining even parity in this representation. We had included the equality operator (which is implemented for all Push types) because it has general utility, for example in bounding loops, but in retrospect we realize that the equality instruction, when applied to the `BOOLEAN` type, provides a Boolean function that fundamentally transforms the parity problem and that was not in Koza's function set. We therefore ran our tests again without including equality instructions in the instruction set. We removed all equality instructions rather than just the equality instruction for the `BOOLEAN` type since Boolean equality could easily be constructed from other forms of equality in conjunction with the `CONVERT` instruction.

Table 6 shows the results of the runs without equality instructions. (This table again represents the results of 100 runs in each condition, with the parameters listed in Table 4.) As expected, the problem without equality is more difficult and we did not find a significant number of solutions in the initial random populations. (We found only one, in a run on even-3-parity.) Again, as the arity of the problems increase, the percentage of solutions that appear to use modularity increases. In addition, while the overall performance of PushGP suffered without the equality instructions the trend with respect to scaling was maintained: in comparison to

Table 6. Results of PushGP runs on even-parity problems without equality instructions

Arity	Effort	% using IF	% using `DO or DO*`	Effort relative to Koza without ADFs	Effort relative to Koza with ADFs
3	1,440,000	100	38	15X	22.5X
4	3,360,000	100	50	8.75X	19X
5	7,392,000	82	69	1.1X	16X
6	9,216,000	83	62	0.13X	6.9X

Note. Each row was calculated on the basis of 100 independent runs.

Koza's results PushGP still scales better, in terms of computational effort, as the arity of the problems increases.

No longer provided with an equality operator, PushGP solves even-parity problems primarily via the creation of nested if-else structures that respond to the Boolean inputs. The construction of if-else structures appears to become more ponderous as the arity of the problems increase, and at arity five and above programs appear that use addition and subtraction of Boolean values converted into integers, the results of which are fed into list index instructions. In some senses `IF` is yet another Boolean operator, and for reasons like those described above we were interested in the performance of PushGP without *either* equality or `IF`.

Table 7 shows the results of the final experiments in this sequence. These runs were identical to those described previously except that neither the equality instructions nor `IF` were included in the function set. While PushGP requires a very large amount of effort to solve the even-3-parity problem, the amount of increase for higher arities is very low. Again, we see excellent scaling behavior that significantly outpaces standard genetic programming even with ADFs. Interestingly, even though the `IF` instruction was used extensively in the first set of experiments without the equality instruction, implying that its presence was useful in solving even parity, the computational effort actually improved after removing *both* equality and `IF` for arities above 3.

5. Autoconstructive evolution with Pushpop

We call our Push-based autoconstructive evolution system *Pushpop*, for *Push* program *pop*ulation. Our current version of Pushpop inherits the skeleton of its algorithm from traditional genetic programming—it is a generation-based algorithm in which individuals are executed sequentially and in which the higher-performance

Table 7. Results of PushGP runs on even-parity problems without equality instructions or `IF`

Arity	Effort	% Using `DO or DO*`	Effort relative to runs without = but with `IF`	Effort relative to Koza without ADFs	Effort relative to Koza with ADFs
3	3,072,000	92	2.1X	32X	48X
4	3,312,000	96	0.98X	8.6X	18X
5	3,696,000	95	0.5X	0.56X	8X
6	5,760,000	98	0.6X	0.08X	4.2X

Note. Each row was calculated on the basis of 100 independent runs.

individuals are allowed to contribute more children to the following generation. This is not essential to the concept of autoconstructive evolution, however, and one could easily imagine a version of Pushpop which instead used a Tierra-like scheme of time-slicing the execution of all of the individuals in a population. As in Tierra, reproduction could happen during the execution of the individuals and without synchronization to a generational clock. Natural selection would then operate not only on the basis of problem-solving performance (enforced by a "reaper" as in Tierra) but also on the basis of replication efficiency.

Indeed, we are considering development of such a version Pushpop at some point in the future. For the sake of simplicity, however, our first version sticks as close as possible the basic genetic programming algorithm, deviating only where necessary to implement autoconstructive evolution.

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

Children are produced by the individual programs as they are evaluated for fitness, so there is no role for the standard genetic operators of mutation, crossover, and reproduction. We must nonetheless filter the children that will be admitted to the next generation, 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 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.

Various mate selection strategies are enabled by these instructions. Since the code of the "mate" is pushed onto the running program's stack it is available for inspection using generic expression-manipulation instructions; this enables genotype-based mate selection strategies beyond those directly supported by `OTHER-TAG`. Phenotype-based mate selection strategies are less richly supported but some are available through the tournament mechanisms underlying `ELDER` and `OTHER`. Because distance in the population and relatedness are correlated, `NEIGHBOR` allows for relatedness-based mate selection strategies.

5.4. Diversity management

As Edmonds noted in his prior work it can be difficult to maintain diversity in a system with evolving genetic operators [12]. 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.

We have found these syntactic diversity constraints to be necessary in almost all circumstances to avoid premature convergence. Usually they suffice, although they do not rule out the possibility of programs that produce functionally identical children that vary only in some small piece of unexecuted code. If this form of semantic convergence becomes problematic then semantic diversity can be encouraged through a variety of mechanisms. One form of semantic diversity constraint we have used 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 Pushpop run the system must first achieve diversifying recursive replication; in other words the system must produce (via random search) 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 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 and since there are many qualitatively different ways to achieve reproductive competence with the Push instruction set.

5.6. *Results*

Pushpop has been run on a range of symbolic regression problems [18], 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. Often, however, a stable (though diverse) equilibrium is reached before the target problem is solved.

Sub-optimal equilibria 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" [47]. 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* (a technique related to *dynamic subset selection* [13]), 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 $n/2$ in the first generation, cases 2 through $n/2+1$ 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.

Although our work on the biological implications of Pushpop is still at an early stage we believe that the evolutionary dynamics of Pushpop will provide a rich source of data. In previous work we described anecdotal observations of transitions between sexual and asexual reproduction strategies in successful Pushpop runs and speculated about their connections to biological phenomena [38]. In more recent work we have examined the diversity of successful parents over the course of a successful run and noted certain recurring patterns. For example, as shown in Figure 1

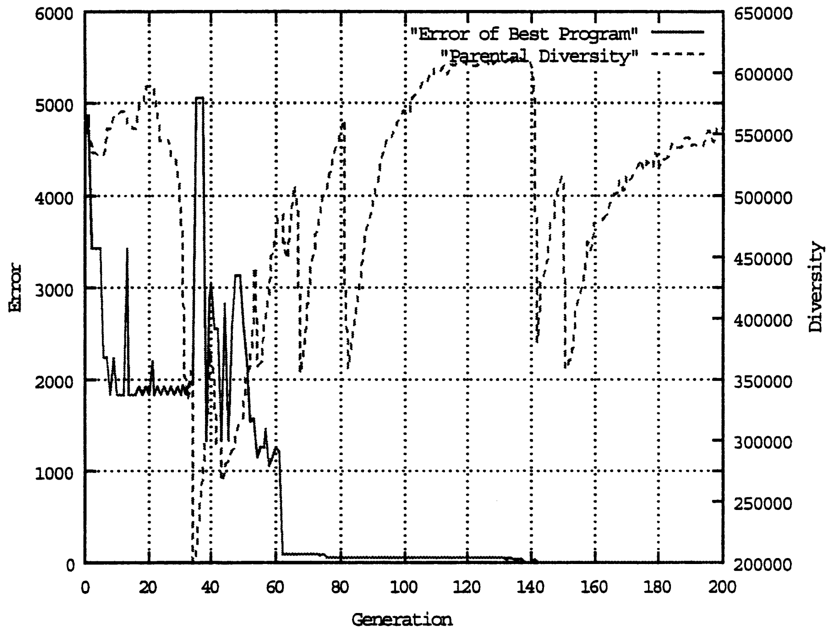

Figure 1. Error of the best program and diversity of successful parents over the course of a Pushpop run on a symbolic regression problem with the target function $y = 5x^2 + x - 2$. Diversity of two individuals was calculated as the sum, over all unique expressions in either of the individuals, of the difference between the number of occurrences of the expression in the two individuals. The graphed diversity measure is the sum of the diversities of all pairs of individuals in a randomly selected set of 128 successful parents from each generation. When less than 128 parents were successful (in some generations before reproductive competence, which occurred here at generation 32) the graph repeats the value from the previous generation. This run used a population size of 2048, a tournament size of 32, 16 fitness cases (0–15), and a maximum program size of 64 points.

we sometimes see diversity climb during periods in which there is no improvement in fitness (error), followed by fitness improvements which then trigger dips in diversity. These data provide a new window into connections between diversity and adaptation such as those studied with Avida [49]. In Avida one manually sets the mutation rate and can then observe the effects on adaptation (including "survival of the flattest"), but in Pushpop diversification is endogenous and the effective mutation rate (the diversity) is a dynamic, emergent feature of the evolving system. This allows one to explore a host of new questions about the stability of diversification and about the effects that different diversification patterns have on adaptation.

6. Conclusions

The Push programming language has been described in detail and 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; we presented several examples demonstrating the power and flexibility of PushGP and initial data suggesting that it can automatically discover good modularizations and thereby scale up to complex problems. 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.

Source code is available for research versions of the Push interpreter and PushGP. This code is available for non-commercial educational and research uses only; see http://hampshire.edu/lspector/push.html.

Appendix: Push language reference

The documentation here aims to convey the essential features of the language but it cannot be 100% complete—for full details please consult the source code that can be found at http://hampshire.edu/lspector/push.html.

Types and instructions

This section documents the Push types and instructions that were implemented at the time of this writing. The language can be easily extended and this set of types and instructions should not be considered complete or final. Conversely, the entire language need not be used for any particular run of PushGP or Pushpop.

Unless otherwise noted all arguments used by instructions are popped from their stacks. Each type inherits all instructions from its parent and ancestors, with locally defined instructions taking precedence. The type to which an instruction will be applied is determined by consulting (but not popping) the `TYPE` stack.

- Type: PUSH-BASE-TYPE (no parent)
  Instructions:

  DUP Push a duplicate of the top item onto the stack.
  POP Remove the top item of the stack.
  SWAP Swap the top two items of the stack.
  REP Replace the second item with the first (in other words, delete the second item).
  $=$ Pop the top two items and push `T` on the `BOOLEAN` stack if they are equal (or `NIL` otherwise).
  NOOP Do nothing.
  PULL Remove an item from deep in the stack and push it on top; the index of the item is popped from the `INTEGER` stack.
  PULLDUP Like `PULL` but does not remove the pulled item.
  SET Associate the top item with the top `NAME`.
  GET Push the item associated with the top `NAME`.
  CONVERT Convert a value from the type that is second on the `TYPE` stack to the type that is first on the `TYPE` stack.
- Type: NUMBER (parent: PUSH-BASE-TYPE)
  Instructions:

  $+$ Push the sum of the top two items.
  $-$ Push the difference of the top two items.
  $*$ Push the product of the top two items.
  / Push the quotient of the top two items; division by zero produces zero.
  $<$ Push `T` on the `BOOLEAN` stack if the second item is less than the first item (or `NIL` otherwise).
  $>$ Push `T` on the `BOOLEAN` stack if the second item is greater than the first item (or `NIL` otherwise).
- Type: INTEGER (parent: NUMBER)
  Instructions:

  RAND Push a random integer onto the stack; the range is user configurable.
  PULL Like the more generic `PULL` but more complicated since the index must come from the target stack (the index is the first popped value).
  PULLDUP Like the more generic `PULLDUP` but more complicated since the index must come from the target stack (the index is the first popped value).
  / Like the more generic / but truncates the result to produce and integer.
- Type: FLOAT (parent: NUMBER)
  Instructions:

  RAND Push a random floating point number onto the stack; the range is user configurable.
- Type: BOOLEAN (parent: PUSH-BASE-TYPE)
  Instructions:

  AND Push the result of the logical AND of the top two items.

OR Push the result of the logical OR of the top two items.
NOT Push the result of the logical NOT of the top item.
NAND Push the result of the logical NAND of the top two items.
NOR Push the result of the logical NOR of the top two items.

- Type: TYPE (parent: PUSH-BASE-TYPE)
  Instructions: (none)
- Type: NAME (parent: PUSH-BASE-TYPE)
  Instructions:
  RAND Push a new, unique name onto the stack.
- Type: EXPRESSION (parent: PUSH-BASE-TYPE)
  Instructions: (Note for all `EXPRESSION` instructions: if an argument is supposed to be a list but is instead an atom then a list containing the atom is used instead.)
  QUOTE Push the next form passed to the evaluator onto the stack without evaluation.
  ATOM Push `T` onto the `BOOLEAN` stack if the top item is an atom (not surrounded by parentheses), or `NIL` otherwise.
  NULL Push `T` onto the `BOOLEAN` stack if the top item is an empty list, or `NIL` otherwise.
  CAR Push the first element of the top item onto the stack.
  CDR Push the tail (all but the first element) of the top item onto the stack.
  CONS Push a list consisting of the second item added to the front of the first item.
  LIST Push a list consisting of the second and first items.
  APPEND Push a list formed from the contents of the second and first items.
  NTH Push the element of the top item (a list) indexed by the item on top of the `INTEGER` stack. Indexing is zero-based and is taken modulo the length of the list.
  NTHCDR Like `NTH` but pushes the tail of the list so indexed.
  MEMBER Push `T` onto the `BOOLEAN` stack if the second item occurs within the first at the top level (and `NIL` otherwise).
  POSITION Push the position of the second item within the first item onto the `INTEGER` stack; pushes -1 if no match is found.
  SUBST Push the result of substituting the third item for the second item in the first item.
  RAND Push a random expression; the `INTEGER` stack provides a size limit (in points).
  CONTAINS Push `T` onto the `BOOLEAN` stack if the second item occurs within the first at the any level (and `NIL` otherwise).
  CONTAINER If `CONTAINS` would push `T` then push the expression that immediately encloses the match; otherwise push `NIL` (onto the relevant `EXPRESSION` stack).
  PERTURB Push a stochastically perturbed copy of the top item. Perturbation will only replace atoms with other atoms. The degree of perturbation is controlled by an integer which is popped from the `INTEGER` stack. A value of 0 for the integer results in no perturbation. A value of n for $n \neq 0$ means that each atom has an $abs(\frac{1}{n})$ probability of being perturbed. This provides a powerful tool for building mutation operators.

REPLACE-ATOMS Push a copy of the top of item in which the atoms have been replaced with the atoms from the second item (taken in order).

MIX-ATOMS Push a copy of the top of item in which the atoms have been replaced with the atoms from the first and second items, with each atom being chosen from one or the other item with a 50% probability. This provides a powerful tool for building "uniform crossover" operators.

OTHER For Pushpop only. Push the code of another individual in the population. See text for details.

ELDER For Pushpop only. Push the code of another individual in the population. See text for details.

NEIGHBOR For Pushpop only. Push the code of another individual in the population. See text for details.

OTHER-TAG For Pushpop only. Push the code of another individual in the population. See text for details.

INSERT Push the result of inserting the second item into the first item. The depth-first-traversal index of the insertion point is taken from the `INTEGER` stack, modulo the number of points in the expression into which the insertion is being made.

EXTRACT Push the subexpression of the top item with depth-first-traversal index obtained from the `INTEGER` stack.

INSTRUCTIONS Push a list of the instructions that are implemented for the type on top of the `TYPE` stack.

LENGTH Push the length of the top item (counting only top-level elements).

SIZE Push the number of points in the top item onto the `INTEGER` stack.

DISCREPANCY Push a measure of the difference between the top two items onto the `INTEGER` stack. This is calculated as the sum, over all unique expressions in either of the individuals, of the difference between the number of occurrences of the expression in the two individuals.

- Type: CODE (parent: EXPRESSION)
  Instructions:

DO Recursively invoke the interpreter on the top item. After evaluation the stack is popped; normally this pops the expression that was just evaluated, but if the expression itself manipulates the stack then this final pop may end up popping something else.

DO* Like `DO` but pop the expression before evaluating it.

IF If the top item of the `BOOLEAN` stack is true then recursively evaluate the second item; otherwise recursively evaluate the first item. In either case pop both items (and the `BOOLEAN` value).

MAP Iterate the second item once for each element in the first item (a list). First pop both items, then do the iterations (each time first pushing the appropriate element) and collect all results (top items after execution) into a new list; push this list after iteration is complete.

- Type: CHILD (parent: EXPRESSION)
  Instructions: (none)

The type stack bottom

For each instruction execution the TYPE stack must be consulted to determine the type upon which the instruction will operate. To ensure that a method can always be chosen for any instruction the following pre-defined "type stack bottom" is appended to the TYPE stack whenever it is consulted:

(top)
INTEGER
BOOLEAN
CODE
CHILD
TYPE
NAME
(bottom)

Type conversion

The following listing specifies how values are converted between all pairs of currently implemented types:

- INTEGER ⇒ INTEGER: The same value.
- FLOAT ⇒ INTEGER: Truncation.
- BOOLEAN ⇒ INTEGER: T ⇒ 1, NIL ⇒ 0.
- TYPE ⇒ INTEGER: Position of the type in the list of all types (which are maintained in the order that they were defined).
- NAME ⇒ INTEGER: 0.
- CODE ⇒ INTEGER: The top-level length of the expression.
- CHILD ⇒ INTEGER: The top-level length of the expression.
- INTEGER ⇒ FLOAT: The same value as a floating-point number.
- FLOAT ⇒ FLOAT: The same value.
- BOOLEAN ⇒ FLOAT: T ⇒ 1.0, NIL ⇒ 0.0.
- TYPE ⇒ FLOAT: Position of the type in the list of all types (which are maintained in the order that they were defined) as a floating-point number.
- NAME ⇒ FLOAT: 0.0.
- CODE ⇒ FLOAT: The top-level length of the expression as a floating-point number.
- CHILD ⇒ FLOAT: The top-level length of the expression as a floating-point number.
- INTEGER ⇒ BOOLEAN: 0 ⇒ NIL, otherwise T.
- FLOAT ⇒ BOOLEAN: 0.0 ⇒ NIL, otherwise T.
- BOOLEAN ⇒ BOOLEAN: The same value.
- TYPE ⇒ BOOLEAN: NIL.
- NAME ⇒ BOOLEAN: NIL.
- CODE ⇒ BOOLEAN: Empty expression ⇒ NIL, otherwise T.
- CHILD ⇒ BOOLEAN: Empty expression ⇒ NIL, otherwise T.

- `INTEGER` ⇒ `TYPE`: `INTEGER`.
- `FLOAT` ⇒ `TYPE`: `FLOAT`.
- `BOOLEAN` ⇒ `TYPE`: `BOOLEAN`.
- `TYPE` ⇒ `TYPE`: `TYPE`.
- `NAME` ⇒ `TYPE`: `NAME`.
- `CODE` ⇒ `TYPE`: `CODE`.
- `CHILD` ⇒ `TYPE`: `CHILD`.
- `INTEGER` ⇒ `NAME`: `INTEGER`.
- `FLOAT` ⇒ `NAME`: `FLOAT`.
- `BOOLEAN` ⇒ `NAME`: `BOOLEAN`.
- `TYPE` ⇒ `NAME`: `TYPE`.
- `NAME` ⇒ `NAME`: `NAME`.
- `CODE` ⇒ `NAME`: `CODE`.
- `CHILD` ⇒ `NAME`: `CHILD`.
- Anything ⇒ `CODE`: The value, quoted literally.
- Anything ⇒ `CHILD`: The value, quoted literally.

Acknowledgments

Mark Feinstein, Raymond Coppinger, Steven Gustafson, Maarten Keijzer, Wolfgang Banzhaf, Sean Luke, Chris Perry, Jaime Davila, Christopher Jarvis, Herbert J. Bernstein, Rebecca S. Neimark, and anonymous reviewers provided 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.

Notes

1. An alternative to runtime limits is described in [24].
2. Another recent related system is SeMar [43].
3. All stacks are available as-is to the recursively executed code, and any changes to the stacks (except changes to the top element of the `CODE` stack, which will be popped upon completion of a call to `DO`) persist after the call to `DO` or `DO*`.
4. This is the same algorithm used to implement the `CODE RAND` Push instruction. Each instruction, constant, or pair of parentheses counts as one "point."
5. In uniform crossover each atom in the child comes from one parent or the other with 50% probability.

6. These examples and the accompanying discussion were first presented in [38]; the text here has been adapted from that paper.
7. The current version of Push is implemented in Lisp and therefore inherits Lisp's conventions for the representation of true and false [15].
8. These parameters were chosen arbitrarily; there is no particular justification for their values and they were not optimized.
9. If the item on top of the appropriate expression stack is just an atom then a list containing just that atom is used instead.
10. Again, these parameters were chosen arbitrarily and were not optimized.
11. The algorithm for computing computational effort was developed by Koza and is described on pages 99 through 103 of [19]. Briefly, one conducts a large number of runs with the same parameters (except random seeds) and begins by calculating $P(M, i)$, the cumulative probability of success by generation i using a population of size M. For each generation i this is simply the total number of runs that succeeded on or before the ith generation, divided by the total number of runs conducted. From $P(M, i)$ one can calculate $I(M, i, z)$, the number of individuals that must be processed to produce a solution by generation i with probability greater than z. Following the convention in the literature we use a value of $z = 99\%$. $I(M, i, z)$ can be calculated using the following formula:

$$I(M, i, z) = M * (i + 1) * \left\lceil \frac{\log(1 - z)}{\log(1 - P(M, i))} \right\rceil$$

The more steeply the graph of $I(M, i, z)$ falls, and the lower its minimum, the better the genetic programming system is performing. Koza defines the minimum of $I(M, i, z)$ as the "computational effort" required to solve the problem.

References

1. C. Adami and C. T. Brown, "Evolutionary learning in the 2D artificial life system 'Avida'," in Artificial Life IV, MIT Press: Cambridge, MA, 1995, pp. 377–381.
2. P. J. Angeline, "Adaptive and self-adaptive evolutionary computations," in Computational Intelligence: A Dynamic Systems Perspective, IEEE Press: New York, 1995, pp. 152–163.
3. P. J. Angeline, "Morphogenic evolutionary computations: Introduction issues and examples," in Evolutionary Programming IV: The Fourth Annual Conference on Evolutionary Programming, MIT Press: Cambridge, MA, 1995, pp. 387–401.
4. P. J. Angeline, "Two self-adaptive crossover operators for genetic programming," in Advances in Genetic Programming 2, P. J. Angeline and K. E. Kinnear, Jr. (eds.), MIT Press: Cambridge, MA, 1996, pp. 89–110.
5. P. J. Angeline and J. B. Pollack, "The evolutionary induction of subroutines," in Proc. Fourteenth Ann. Conf. Cognitive Science Society, Lawrence Erlbaum: London, 1992.
6. T. Bäck, "Self-adaptation in genetic algorithms," in Toward a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life, MIT Press: Cambridge, MA, 1992, pp. 263–271.
7. W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone, Genetic Programming: An Introduction, Academic Press/Morgan Kaufmann: New York/Los Altos, CA, 1998.
8. S. Brave, "Evolving recursive programs for tree search," in Advances in Genetic Programming 2, P. J. Angeline and K. E. Kinnear, Jr. (eds.), MIT Press: Cambridge, MA, 1996, pp. 203–220.
9. W. S. Bruce, "Automatic generation of object-oriented programs using genetic programming," in Genetic Programming 1996: Proc. First Ann. Conf., J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo (eds.), MIT Press: Cambridge, MA, 1996, pp. 267–272.
10. W. S. Bruce, "The lawnmower problem revisited: Stack-based genetic programming and automatically defined functions," in Genetic Programming 1997: Proc. Second Ann. Conf., J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo (eds.), Morgan Kaufmann: Los Altos, CA, 1997, pp. 52–57.

11. P. Dittrich and W. Banzhaf, "Self-evolution in a constructive binary string system," Artificial Life, vol. 4, pp. 203–220, 1998.
12. B. Edmonds, "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, 1998.
13. C. Gathercole, "An investigation of supervised learning in genetic programming," PhD Thesis, University of Edinburgh, 1998.
14. P. Graham, On LISP: Advanced Techniques for Common LISP, Prentice-Hall: Englewood Cliffs, NJ, 1993.
15. P. Graham, ANSI Common Lisp. Prentice-Hall: Englewood Cliffs, NJ, 1996.
16. W. E. Hart, "A convergence analysis of unconstrained and bound constrained evolutionary pattern search," Evolutionary Computation, vol. 9(1), pp. 1–23, 2000.
17. W. Kantschik, P. Dittrich, M. Brameier, and W. Banzhaf, "MetaEvolution in graph GP," Proc. EuroGP'99, LNCS, vol. 1598. Springer-Verlag: Berlin, 1999, pp. 15–28.
18. J. R. Koza, Genetic Programming: On the Programming of Computers by Means of Natural Selection, MIT Press: Cambridge, MA, 1992.
19. J. R. Koza, Genetic Programming II: Automatic Discovery of Reusable Programs, MIT Press: Cambridge, MA, 1994.
20. J. R. Koza, D. Andre, F. H. Bennett III, and M. Keane, Genetic Programming 3: Darwinian Invention and Problem Solving, Morgan Kaufmann: Los Altos, CA, 1999.
21. W. B. Langdon, Data Structures and Genetic Programming: Genetic Programming + Data Structures = Automatic Programming!, Kluwer: Dordrecht, 1998.
22. R. E. Lenski, C. Ofria, T. C. Collier, and C. Adami, "Genome complexity, robustness and genetic interactions in digital organisms," Nature, vol. 400, pp. 661–664, 1999.
23. L. Margulis, D. Sagan, and N. Eldredge, What is Life?, University of California Press: Berkeley, CA, 2000.
24. S. R. Maxwell III, "Experiments with a coroutine model for genetic programming," in Proc. 1994 IEEE World Congress on Computational Intelligence, IEEE Press: New York, 1994, pp. 413–417.
25. D. J. Montana, "Strongly typed genetic programming," Evolutionary Computation, vol. 3, no. 2, pp. 199–230, 1995.
26. P. Nordin, W. Banzhaf, and F. D. Francone, "Efficient evolution of machine code for CISC architectures using instruction blocks and homologous crossover," in Advances in Genetic Programming 3, L. Spector, W. B. Langdon. U.-M. O'Reilly, and P. J. Angeline (eds.), MIT Press: Cambridge, MA, 1999, pp. 275–299.
27. T. R. Osborn, A. Charif, R. Lamas, and E. Dubossarsky, "Genetic logic programming," in IEEE Conf. Evolutionary Comput., vol. 2, IEEE Press: New York, 1995, pp. 728–732.
28. A. N. Pargellis, "The spontaneous generation of digital life," Physica D, vol. 91, pp. 86–96, 1996.
29. W. Pedrycz and M. Reformat, "Evolutionary optimization of logic-oriented systems," in Proc. Genetic and Evolutionary Comput. Conf. (GECCO-2001), 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 (eds.), Morgan Kaufmann: Los Altos, CA, 2001, pp. 1389–1396.
30. T. Perkis, "Stack-based genetic programming," in Proc. 1994 IEEE World Congress on Comput. Intell., IEEE Press: New York, 1994, pp. 148–153.
31. T. S. Ray, "Is it alive or is it GA?," in Proc. Fourth Inter. Conf. Genetic Algorithms, Morgan Kaufmann: Los Altos, CA, 1991, pp. 527–534.
32. A. Robinson, "Genetic programming: Theory, implementation, and the evolution of unconstrained solutions," Hampshire College Division III (senior) thesis. http://hampshire.edu/lspector/robinson-div3.pdf, 2001.
33. J. P. Rosca and D. H. Ballard, "Discovery of subroutines in genetic programming," in Advances in Genetic Programming 2, P. J. Angeline and K. E. Kinnear, Jr. (eds.), MIT Press: Cambridge, MA, 1996, pp. 177–202.
34. W. P. Salman, O. Tisserand, and B. Toulot, FORTH, Springer-Verlag: Berlin, 1984.
35. M. Sipper, "Fifty years of research on self-replication: An overview," Artificial Life, vol. 4, no. 3, pp. 237–257, 1998.
36. M. Sipper and J. Reggia, "Go forth and replicate," Scientific American, August, 2001.

37. L. Spector, "Simultaneous evolution of programs and their control structures," in Advances in Genetic Programming 2, P. J. Angeline and K. E. Kinnear, Jr. (eds.), MIT Press: Cambridge, MA, 1996, pp. 137–154.
38. L. Spector, "Autoconstructive evolution: Push, PushGP, and Pushpop," in Proc. Genetic and Evolutionary Comput. Conf., GECCO-2001, 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 (eds.), Morgan Kaufmann: Los Altos, CA, 2001, pp. 137–146.
39. L. Spector and K. Stoffel, "Ontogenetic programming," in Genetic Programming 1996: Proc. First Ann. Conf., J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo (eds.), MIT Press: Cambridge, MA, 1996, pp. 394–399.
40. L. Spector and K. Stoffel, "Automatic generation of adaptive programs," in From Animals to Animats 4: Proc. Fourth Inter. Conf. Simulation of Adaptive Behavior, MIT Press: Cambridge, MA, 1996, pp. 476–483.
41. C. R. Stephens, I. G. Olmedo, J. M. Vargas, and H. Waelbroeck, "Self-adaptation in evolving systems," Artificial Life, vol. 4, pp. 183–201, 1998.
42. K. Stoffel and L. Spector, "High-performance, parallel, stack-based genetic programming," in Genetic Programming 1996: Proc. First Annual Conf., J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo (eds.), MIT Press: Cambridge, MA, 1996, pp. 224–229.
43. H. Suzuki, "Evolution of self-reproducing programs in a core propelled by parallel protein execution," Artificial Life, vol. 6, no. 2, pp. 103–108, 2000.
44. E. Tchernev, "Forth crossover is not a macromutation?," in Genetic Programming 1998: Proc. Third Ann. Conf., 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 (eds.), Morgan Kaufmann: Los Altos, CA, 1998, pp. 381–386.
45. A. Teller, "Evolving programmers: The co-evolution of intelligent re-combination operators," in Advances in Genetic Programming 2, P. J. Angeline and K. E. Kinnear, Jr. (eds.), MIT Press: Cambridge, MA, 1996, pp. 45–68.
46. E. Tunstel and M. Jamshidi, "On genetic programming of fuzzy rule-based systems for intelligent control," Inter. J. Intelligent Automation and Soft Computing, vol. 2, no. 3, pp. 273–284, 1996.
47. P. D. Turney, "A simple model of unbounded evolutionary versatility as a largest-scale trend in organismal evolution," Artificial Life, vol. 6, no. 2, pp. 109–128, 2000.
48. P. Walsh, "Evolving pure functional programs," in Genetic Programming 1998: Proc. Third Ann. Conf., Morgan Kaufmann: Los Altos, CA, 1998, pp. 399–402.
49. C. O. Wilke, J. L. Wang, C. Ofria, R. E. Lenski, and C. Adami, "Evolution of digital organisms at high mutation rates leads to survival of the flattest," Nature, vol. 412, pp. 331–333, 2001.
50. G. T. Yu, "An analysis of the impact of functional programming techniques on genetic programming," PhD Thesis, University College, London, 1999.