Chapter 2

TOWARDS PRACTICAL AUTOCONSTRUCTIVE EVOLUTION: SELF-EVOLUTION OF PROBLEM-SOLVING GENETIC PROGRAMMING SYSTEMS

Lee Spector
Cognitive Science, Hampshire College, Amherst, MA, 01002-3359 USA.

Abstract Most genetic programming systems use hard-coded genetic operators that are applied according to user-specified parameters. Because it is unlikely that the provided operators or the default parameters will be ideal for all problems or all program representations, practitioners often devote considerable energy to experimentation with alternatives. Attempts to bring choices about operators and parameters under evolutionary control, through self-adaptative algorithms or meta-genetic programming, have been explored in the literature and have produced interesting results. However, no systems based on such principles have yet been demonstrated to have greater practical problem-solving power than the more-standard alternatives. This chapter explores the prospects for extending the practical power of genetic programming through the refinement of an approach called *autoconstructive evolution*, in which the algorithms used for the reproduction and variation of evolving programs are encoded in the programs themselves, and are thereby subject to variation and evolution in tandem with their problem-solving components. We present the motivation for the autoconstructive evolution approach, show how it can be instantiated using the Push programming language, summarize previous results with the *Pushpop* system, outline the more recent *AutoPush* system, and chart a course for future work focused on the production of practical systems that can solve hard problems.

Keywords: genetic programming, meta-genetic programming, autoconstructive evolution, Push, PushGP, Pushpop, AutoPush

1. Introduction

The work described in this chapter is motivated both by features of biological evolution and by the requirements for the high-performance problem-solving systems of the future.

Under common conceptions of biological evolution the variation of genotypes from parents to children, and hence the diversification of phenotypes from progenitors to their descendants, is essentially random prior to selection. Offspring vary randomly, it is said, and selection acts on the resulting diversity by allowing the better-adapted random variants to survive and reproduce. Such conceptions are held not only by the lay public but also by theorists such as Jerry Fodor and Massimo Piattelli-Palmarini who, in their book *What Darwin Got Wrong,* criticize Darwinian theory in part on the grounds that the random "generate and test" algorithm at its core is insufficiently powerful to account for the facts of natural history (Fodor and Piattelli-Palmarini, 2010).

But diversification in nature, while certainly random in some respects, is also clearly non-random in several others. If one were to modify DNA molecules in truly random ways, considering all chemical bonds to be equally good candidates for breakage and re-connection, then one would not end up with DNA molecules at all but instead with some other sort of organic soup. Cellular machinery copies DNA, and repairs copying errors, in ways that allow for certain kinds of "errors" but only within tightly constrained bounds. At higher levels of organization variation is constrained by genetic regulatory processes, the mechanics of sexual recombination, cell division and development, and, at a much higher level of organization, by social structures that guide non-random mate selection. All of these constraints emerge from reproductive processes that have themselves evolved over the course of natural history. There is a large literature on such constraints, including a recent theory of "facilitated variation" (Gerhart and Kirschner, 2007), and summaries of the evolution of variation from pre-biotic Earth to the present (Maynard Smith and Szathmáry, 1999).

Whether or not the evolved-non-randomness of biological variation constitutes a significant critique of neo-Darwinism or of the historical Darwin, as claimed by Fodor and Piattelli-Palmarini, is beyond the scope of the present discussion. For our purposes, however, two related points should be made. First, while truly random variation, filtered by selection, may be too weak of a mechanism to have produced the sequence of phenotypes observed over time in the historical record, it is possible for random variation, *when acting on the reproductive mechanisms themselves,* to produce variation mechanisms that are *not* purely random. This is presumably what happened in natural history. Second, this bootstrapping process, of the evolution of adaptive, not-entirely-random variation by means of the initially random variation *of the variation* *mechanisms*, might also be applied to evolutionary problem-solving technologies.

Why would we want to do this? One reason is that the problem-solving power of current evolutionary computing technologies is limited by the nature of the variation mechanisms that we build into these systems by hand. Consider, for example, the standard mutation operators used in genetic programming. Subtree replacement, applied uniformly to the nodes in a program tree (or uniformly to interior vs. leaf nodes with a specified probability), involving the replacement of subtrees with newly-generated random subtrees, provides a form of variation that leads to solutions in some but not all problem environments. This has led to the development of a wide range of alternative mutation operators; see, for example, the "Mutation Cookbook" section of (Poli et al., 2008, pp. 42–44). But which of these will be most helpful in which circumstances, and which others, perhaps not yet invented, may be needed to solve challenging new problems?

The field currently has no satisfying answer to this question, which will become all the more pressing as genetic programming systems incorporate more expressive and heterogeneous program representations. In the context of such representations it may well make sense for different program elements or program locations to have different variation rates or procedures, and it will not be obvious, in advance, how to make these choices. The question will also become all the more pressing as genetic programming systems are applied to ever more complex problems, about which the system designers will have less knowledge and intuition. And the question will be raised with even greater urgency with respect to recombination operators such as crossover, for which there even more open questions (e.g. about how to choose crossover partners) that currently require the user to make choices that may not be optimal.

Two approaches to these general issues that have previously been explored in the literature are "self-adaptation" and "meta-genetic programming." Many forms of self-adaptation have been investigated, both within genetic programming and in other areas of evolutionary computation (with many examples including (Angeline, 1995; Spears, 1995; Angeline, 1996; Eiben et al., 1999; MacCallum, 2003; Fry et al., 2005; Beyer and Meyer-Nieberg, 2006; Vafaee et al., 2008; Silva and Dignum, 2009)). In all of these systems the parameters of the evolutionary algorithm are varied and subjected to some form of selection, whether the variation and selection is accomplished by means of the overarching evolutionary algorithm, by a secondary evolutionary algorithm, or by some other machine learning technique. In some cases the parameters are adapted on an individual basis, while in others the self-adaptive system modifies global parameters that apply to an entire population. In general, however, these systems vary only pre-selected parameters of the variation operators in pre-specified ways, and they do not allow for the evolution of arbitrary methods of variation.

By contrast, the "meta-genetic programming" approach leverages the program-space search capabilities of genetic programming to search for variation operators—which are, after all, themselves programs—during the search for problem-solving programs (Schmidhuber, 1987; Kantschik et al., 1999; Edmonds, 2001; Tavares et al., 2004; Diosan and Oltean, 2009). These systems would appear to have more potential to evolve adaptive variation algorithms, but they have generally been subject to one or both of the following two significant limitations:

- The evolving genetic operators are not associated with specific evolving problem-solving programs; they are expected to apply to all evolving problem-solving programs equally well.
- The evolving genetic operators are restricted to being compositions of a small number of pre-designed components; many conceivable genetic operators will not be representable using these components.

The first of these limitations contrasts with some of the self-adaptive evolutionary algorithms mentioned previously, in which the values of parameters for genetic operators are encoded in individuals. That this "global" conception of the applicability of genetic operators might be a limitation should be evident from a cursory examination of the diversity of reproductive strategies in nature. For example, the reproductive strategies of the dandelion are quite different from those of the tiger, the oyster mushroom, and *Escherichia coli*; nobody would expect the strategies of any of these organisms to work particularly well for any of the others. Of course the diversity present in the Earth's biosphere dwarfs that of any current genetic programming system, but it would nonetheless be quite surprising if the same genetic operators worked equally well across a genetic programming population with any significant diversity. One could well imagine, for example, that a subset of the population might share one particular subtree in which a high degree of mutation is adaptive and a second subtree in which mutation is always deleterious. Other individuals in the population might lack either or both of these subtrees, or they might contain additional code that changes the effects of mutations within these particular subtrees.

The second of these limitations is probably mostly a reflection of the fact that most genetic programming representations limit the expressiveness of the programs that they can evolve more generally. Although several Turing complete representations have been described (for example, (Teller, 1994; Nordin and Banzhaf, 1995; Spector and Robinson, 2002a; Woodward, 2003; Yabuki and Iba, 2004; Langdon and Poli, 2006)), such representations are relatively rare and representations that can easily perform arbitrary transformations on variable-sized programs are rarer still. Nature appears to be quite flexible and inventive in the variation mechanisms that it employs (e.g., mechanisms involving gene duplication), and we can easily imagine cases in which genetic programming systems would benefit from the use of genetic operators that are not simple compositions of hand-designed operator components.

Another line of research that bears on the approach presented here generally appears in the artificial life literature. Systems such as Tierra (Ray, 1991), Avida (Ofria and Wilke, 2004), and SeMar (Suzuki, 2004) all involve the evolution of programs that are partially responsible for their own reproduction, and in which the reproductive mechanisms (including genetic operators) are therefore subject to variation and selection. However, in these systems diversification is generally driven by hand-designed "ancestor" replicators and/or by the effects of hand-designed mutation algorithms that are applied automatically to the results of all code manipulation operations. Furthermore, while some of these systems have been used to solve computational problems their problem-solving power has been quite limited; they have been used to evolve simple logic gates and arithmetic functions, but they have not been applied to the kinds of difficult problems that genetic programming practitioners are interested in solving. This is not surprising, as these systems have generally been developed primarily to study biological evolution, not to solve difficult computational problems.

Additional related work has been conducted in the context of evolved self-reproduction (Taylor, 1999; Sipper and Reggia, 2001) although most of this work has been focused on the evolution of exact replication rather than the evolution of adaptive variation. An exception, and the closest work to that described below, is Koza's work on the "Spontaneous Emergence of Self-Replicating and Evolutionarily Self-Improving Computer Programs" (Koza, 1994). In that work Koza evolved programs that simultaneously solved problems (albeit simple Boolean problems) and produced variant offspring using template-based code self-modification in a "sea" or "Turing gas" of programs (Fontana, 1992).

This chapter describes an approach to self-adaptive genetic programming, called *autoconstructive evolution*, that combines several features of the approaches described above, with the long-term goal of producing a new generation of powerful problem solving systems. The potential advantage of the autoconstructive evolution approach is that it will allow variation mechanisms to co-evolve with the programs to which they are applied, thereby allowing the evolutionary system itself to adapt to its problem environments in significant ways. The autoconstructive evolution approach was first described in 2001 and 2002 (Spector, 2001; Spector, 2002; Spector and Robinson, 2002a; Spector and Robinson, 2002b), using the Pushpop system that leveraged features of the Push programming language for evolved programs. In the next section this earlier work is briefly described. The subsequent section describes more recent work on the approach, using better technology and a more explicit focus on the goal of high performance problem solving, implemented in a newer system called *AutoPush*. The final section of the chapter offers some brief conclusions.

2. Push and Pushpop

An *autoconstructive evolution* system was defined in (Spector and Robinson, 2002a) as "any evolutionary computation system that adaptively constructs its own mechanisms of reproduction and diversification as it runs." In the context of the present discussion, however, that definition is too general, and a more specific definition that captures both the past and present usage would be "any genetic programming system in which the methods for reproduction and diversification are encoded in the individual programs themselves, and are thereby subject to variation and evolution." The goal in the previous work, as in the work described here, is for the ways in which children are produced to be evolved along with the programs to which they will be applied. This is done by encoding the mechanisms for reproduction and diversification *within* the programs themselves, which must be capable of producing children and, in principle, of solving the problem to which the genetic programming system is being applied. The space of possible reproduction and diversification methods is vast and an ideal system would allow evolving programs to reach new and uncharted reaches of this space. Human-designed diversification mechanisms, including human-designed genetic operators, human-specified automatic mutation during code-manipulation, and human-written ancestor programs, should all be avoided.

Of course it will generally be necessary for *some* features of any evolutionary system to be pre-specified; for example, all of the systems described here borrow several pre-specified elements of traditional genetic programming systems, including a generation-based evolutionary loop, a fixed-size population, and tournament selection with a pre-specified tournament size. The focus here is on the evolution of the means by which children are produced from parents, and it is this task for which we currently seek autoconstructive methods.

A prerequisite for this approach is a program representation in which problem-solving functions and child-production functions can both be easily expressed. The *Push* programming language was originally designed specifically for this purpose (Spector, 2001). Push is a stack-based language roughly in the tradition of Forth, but for which each data type has its own stack. Instructions generally take their arguments from the appropriate stacks and push their results onto the appropriate stacks.[1] If an instruction requires arguments that are not present on the appropriate stacks when it is called then it does nothing (it acts as a "no-op").

[1]Exceptions are instructions that draw their inputs from external data structures, for example instructions that access inputs, and instructions that act on external data structures, for example "developmental" instructions that add components to externally-developing representations of circuits or other structured objects.

These specifications mean that even though multiple data types may be present in a program no instruction will ever be called on arguments of the wrong type, regardless of its syntactic position in the program. Among other benefits, this means that there are essentially no syntax constraints on Push programs aside from a requirement that parentheses be balanced. This is particularly useful for systems in which child programs will be produced by evolving programs.

One of Push's most important features for autoconstructive evolution, and for genetic programming more generally, is the fact that "code" is a first-class data type. When a Push program is being executed the code that is queued for execution is stored on a special stack called the "exec" stack, and exec instructions in the program can manipulate the queued instructions in order to implement a wide variety of evolved control structures (Spector et al., 2005). Additional code stacks (including one called simply "code," and in some implementations others with names such as "child") can be used to store and manipulate code for a variety of other purposes. This feature has significant benefits for genetic programming even in a non-autoconstructive context (that is, even when standard, hard-coded genetic operators are used, as in the PushGP system), but here we focus on the use of Push for autoconstructive evolution. Space limits prevent full exposition of the Push language here; see (Spector et al., 2005) and the references therein for further details.[2]

The first autoconstructive evolution system built using Push, called Pushpop, can best be understood as an extension of a more-standard genetic programming system such as PushGP. In PushGP, when a program is being tested for fitness on a particular fitness case it is run and then the problem-solving outputs are collected from the relevant data stacks (typically integer or float) and tested for errors; Pushpop does this as well, but it also simultaneously collects a potential child from the child stack. If the problem to which the system is being applied involves n fitness cases then the testing of each program in the population will produce n potential children. In the reproductive phase tournaments are conducted among parents and children are selected randomly from the set of potential children of the winning parents. If there are insufficient children to fill the child population then newly generated random individuals are used.

In Pushpop, as in any autoconstructive evolution system, care must be taken to prevent the takeover of the population by perfect replicators or other pathological replicants. Because there is no automatic mutation in Pushpop a perfect replicator can rapidly fill the population with copies of itself, after which no evolution (and indeed no change at all) will occur. The production of perfect replicators in Push is generally trivial, because programs are pushed onto the code stack prior to execution. For this reason Pushpop includes a "no cloning" rule that specifies that exact clones will not be allowed into the child popula-

[2]See also http://hampshire.edu/lspector/push.html.

tion. Settings are also available that prohibit children that are identical to any of their ancestors or any other individuals in the population. The "no cloning" rule forces programs to diversify in *some* way, but it does not dictate the mode or extent of diversification. The pathology of perfect replicators in nature was presumably overcome with the aid of vast stretches of time and over vast expanses of the Earth, within which perfect replicators may have arisen but later been eliminated when changes occurred to which they could not adapt. Our resources are much more constrained, however, and so we must proactively cull the individuals that we know cannot possibly evolve.

Programs in a Pushpop population can reproduce using evolved forms of multi-parent recombination, accessing other individuals in the population through the use of a variety of instructions provided for this purpose and using them in any computable way to produce their children (Spector and Robinson, 2002a). In fact, evolving Pushpop programs can access *and then execute* code from other individuals in the population, which means that evolved programs may not work correctly when executed outside of the populations within which they evolved. This is unfortunate from the perspective of a practitioner who is primarily interested in producing a program that will solve a particular problem, since the "solution" may require the entire population to work and it may be exceptionally difficult to understand. The mechanisms for population access in Pushpop are also somewhat complex, and the presence of these mechanisms makes it particularly difficult to analyze the performance of the system. For these reasons the new work described here does not allow executing programs to access the other programs in the population; see below for further discussion.

Pushpop is capable of solving simple symbolic regression problems, and it has served as the basis for studies of the evolution of diversification. For example, one study showed that evolving populations that produce adaptive Pushpop programs—that is, programs that actually solve the problems presented to the system—are reliably more diverse than is required by the "no cloning" rule alone (Spector, 2002). But Pushpop's utility as a problem-solving system is limited, and the focus of the Push project in subsequent years has been on more traditional genetic programming systems such as PushGP. PushGP uses traditional genetic operators but the code-manipulation features of Push nonetheless provide benefits, for example by simplifying the evolution of novel control structures and modular architectures.

More recently, however, the use of Push for autoconstructive evolution has been revisited in light of improvements to the Push language (Spector et al., 2005), the availability of substantially faster hardware, and a clarified focus on the long-term potential of autoconstructive evolution to solve problems that cannot be solved with hand-coded diversification mechanisms.

3. Practical Autoconstructive Evolution

AutoPush is a new autoconstructive genetic programming system, a successor to Pushpop built on the more expressive version 3 of the Push programming language and designed with a more explicit focus on problem-solving power. To that end, several sources of inessential complexity in Pushpop have been removed to aid in the analysis of AutoPush runs and their results.

AutoPush, like Pushpop, uses the basic generational loop of a standard genetic programming system and tournament selection with a pre-specified tournament size. Also like Pushpop it uses no pre-specified genetic operators, no ancestor replicators, and no pre-specified, automatic mutation. And like Pushpop it represents its programs in a Turing complete language so that children may be produced from parents by means of any computable function, modulo limits on execution steps or time.

The current version of AutoPush is asexual—that is, parents must construct their children without having access to other programs in the population—because this eliminates the complexity that may not be necessary and it also simplifies analysis. Asexual programs may be run in isolation, both to solve the target problem and to study the range of children that they produce, and it is easy to store all of their ancestors (of which there will be only as many as there have been generations, while each individual in a sexually-reproducing population may have exponentially many ancestors). Future versions of AutoPush may reintroduce the possibility of recombination by reintroducing instructions that provide access to other individuals in the population; it is our intention to explore this option once the dynamics of the asexual version are better understood. It is also worth noting that the role of sex in biological diversification is a subject of considerable debate, and that asexual organisms diversify in complex and significant ways (Barraclough et al., 2003).

The processes by which programs are tested for problem-solving performance and used to produce children also differ between Pushpop and AutoPush. In Pushpop a potential child is produced for each fitness case, during the calculation of the problem-solving answer for that fitness case. This means that the number of children may depend on the number of fitness cases, which complicates analysis and also changes the way that the algorithm will perform on problems with different numbers of fitness cases. By contrast, in AutoPush no children are produced during fitness testing; any code left on the code stack after a fitness-testing run is ignored.[3] Instead, when an individual is selected

[3]In Pushpop a special child stack is used for the production of children because the code stack is needed for the expression of evolved control structures in Push1, in which Pushpop was implemented. AutoPush is implemented in Push3, in which the new exec stack can be used for evolved control structures, freeing up the code stack for child production.

for autoconstructive reproduction in a tournament it is run again, with an input of 0, to produce a child program for the next generation.[4]

The most significant innovation in AutoPush is a new approach to constraints on birth and selection. Pushpop incorporates a "no cloning" rule but AutoPush goes further, adding more constraints on birth and selection to facilitate the evolution of adaptive diversification. Following the lead of meta-genetic programming developers who judged the fitness of evolving operators by "some measure of success in increasing the fitness of the population they operate on" (Edmonds, 2001), AutoPush incorporates factors based on the history of improvement within the ancestry of an individual.

There are many ways in which one might measure "history of improvement" and many ways in which such measurements might be used in an evolutionary algorithm. For example, Smits et al. define "activity" or "potential to improve" as "the sum of the number of moves [in the program search space] that either improved the fitness or neutral moves that resulted in either no change in fitness or a change that was less than a given (dynamic) tolerance limit" (Smits et al., 2010). They use this measure to select candidates for further testing, crossover, and replacement. Additional comments on varieties and measures of self-improvement can be found in (Schmidhuber, 2006).

In AutoPush the history of improvement is a scalar that summarizes the direction of problem-solving performance changes over the individual's ancestry, with greater weight given to more recent changes (see formula below). It would be tempting to use this measure of improvement only in selection, possibly as a second objective—in addition to problem-solving performance—in the context of a multi-objective selection scheme. But this, by itself, would not work well because selection cannot salvage a population that has become overrun by evolutionary "dead-enders" that can never produce improved descendants. Such dead-enders include not only cloners but also programs of several other categories. For example, consider a population full of programs that produce children that vary only in a subexpression that is never executed. This population is just as un-adaptive as a population of cloners, and it will do no good to select among its individuals on any basis whatsoever. Many other, more subtle categories of dead-enders exist, presenting challenges to any evolutionary system that relies only on selection to drive adaptation. The alternative approach taken in AutoPush is to prevent such dead-enders, when they can be detected, from reproducing at all, and to make room in the population for the children of improvers or at least for new random individuals.

[4]The input value of 0 is arbitrary, and an input value is provided only for the minor convenience of avoiding re-definition of the input-pushing instruction. None of this should be significant as long as we are consistent in the ways that we conduct the autoconstructive reproduction runs.

As a result, we place a variety of constraints on birth and selection which act collectively to promote the evolution of adaptive diversification without specifying the form(s) that the actual diversification algorithms will take. More specifically, we conduct selection using tournaments, with comparisons within the tournament set computed as follows:[5]

- Prefer reproductively competent parents: Individuals that were generated by other individuals beat randomly-generated individuals, and individuals that are "grandchildren" beat all others that are not. If both individuals being compared are grandchildren then the lengths of their lineages are not otherwise decisive.

- Prefer parents with non-stagnant lineages: A lineage is considered stagnant if it has persisted for at least some preset number of generations (6 in the experiments described here) and if problem-solving performance has not changed in the most recent half of the lineage.

- Prefer parents with good problem-solving performance: If neither reproductive competence nor lineage stagnation are decisive then select the parent that does a better job on the target problem.

The constraints on birth make use of two auxiliary definitions, for "improvement" and "code discrepancy." Improvement is a measure of how much the problem-solving performance of a lineage has improved, with greater weight being given to the most recent steps in the lineage. We first compute a normalized vector of changes in problem-solving performance, with improvements represented as 1, declines represented as -1, and repeats of the same value represented as 0. The overall improvement value is then calculated as the weighted average of the elements of this vector, with the weights produced by following function (with decay factor $\delta = 0.1$ for the runs described here):

$$w_{g=current-gen} = 1$$
$$w_{g-1} = w_g * (1 - \delta)$$

Code discrepancy is a measure of the difference between two programs, calculated as the sum, over all unique expressions and sub-expressions in either of the programs, of the difference between the numbers of occurrences of the expression in the two programs. In the context of these definitions we can state the constraints on birth as follows:

[5]These constraints, and those mentioned for birth below, are stated using the numerical parameter values that were chosen, more or less arbitrarily, for the runs described here. Other values may perform better, and further study may provide guidance on setting these values or eliminating the parameters altogether.

- Prevent birth from lineages with at least a preset threshold number of ancestors (4 here) and an improvement of less than some preset minimum (0.1 here).
- Prevent birth from lineages with at least a preset threshold number of ancestors (3 here) and constant discrepancy between parent and child in all generations.
- Prevent birth from parents that received disqualifying fitness penalties, e.g. for nontermination or non-production of result values.
- Prevent birth of children with sizes outside of the specified legal range (here 10–100 points).
- Prevent birth of children that are identical to any of their ancestors.
- Prevent birth of children that are identical to potential siblings; for this test the parent program is run a second time to produce an additional child that is used only for this comparison.

4. Preliminary results

While the approach described here has not yet been shown to solve problems that are out of reach of more conventional genetic programming systems—indeed, it is currently weaker than the more-standard PushGP system—it has solved simple problems and produced illuminating data that may help to deepen our understanding.

For example, in one run on a symbolic regression problem with the target function $y = x^3 - 2x^2 - x$ AutoPush found a solution that descended from the following randomly generated program:[6]

```
((code_if (code_noop) boolean_fromfloat (2) integer_fromfloat)
(code_rand integer_rot) exec_swap code_append integer_mult)
```

While it is difficult to tell from inspection how this program works, even for those experienced in reading Push code, the specific code instructions that are included provide clues about how it constructs children. For example, the `code_rand` instruction generates new random code, and the `code_append` instruction combines two pieces of code on the code stack. It is even more revealing to look at the code outputs from several runs of this program. In this case they are all of the form:

```
(RANDOM-INSTRUCTION (code_if (code_noop) boolean_fromfloat
```

[6]Space limitations prevent full description of the run parameters or the instruction set; see (Spector et al., 2005) and the source code at http://hampshire.edu/lspector/gptp10 for more information.

```
(2) integer_fromfloat) (code_rand integer_rot) exec_swap code_append
integer_mult)
```

where "RANDOM-INSTRUCTION" is some particular randomly chosen instruction. So this program's reproductive strategy is merely to add a new, random instruction to the beginning of itself.

This strategy continues for several generations, with several improvements in problem-solving performance, until something new and interesting happens. In the sixth generation a child is produced with a new list added, rather than just a new instruction, and it also has a new reproductive strategy: it adds something new to the beginning of *both* of its top-level lists. In other words, the sixth-generation individual is of this form:

```
(SUB-EXPRESSION-1 SUB-EXPRESSION-2)
```

where each "SUB-EXPRESSION-n" is a different sub-expression, and the seventh-generation children of this program are all of the form:

```
((RANDOM-INSTRUCTION-1 (SUB-EXPRESSION-1))
 (RANDOM-INSTRUCTION-2 (SUB-EXPRESSION-2)))
```

where each "RANDOM-INSTRUCTION-n" is some particular randomly chosen instruction.

One generation later the problem was solved, by the following program:

```
((integer_stackdepth (boolean_and code_map)) (integer_sub
(integer_stackdepth (integer_sub (in (code_wrap (code_if (code_noop)
boolean_fromfloat (2) integer_fromfloat) (code_rand integer_rot)
exec_swap code_append integer_mult))))))
```

This program inherits the altered reproductive strategy of its parent, augmenting both of its primary sub-expressions with new initial instructions in its children.

In the run described above the only available code-manipulation instructions were those in the standard Push specification, which are modeled loosely on Lisp list-manipulation primitives. In some runs, however, we have added a "perturb" instruction that changes symbols and constants in a program to other random symbols or constants with a probability derived from an integer popped from the integer stack. Perturb, which was also used in some Pushpop runs, is itself a powerful mutation operator, but its availability does not dictate if or how or where it will be used; for example, it would be possible for an evolved reproductive strategy to use perturb on only one part of its code, or to use it with different probabilities on different parts of its code, or to use it conditionally or in conjunction with other code-manipulation instructions. With the perturb instruction included we have been able to solve somewhat more difficult problems such as the symbolic regression of $y = x^6 - 2x^4 + x^2 - 2$, and we are actively exploring application to more difficult problems and analysis of the resulting programs and lineages, with the hypothesis that more complex and adaptive reproductive strategies will emerge in the context of more challenging problem environments.

5. Conclusions

The specific results reported here are preliminary, and the hypothesis that autoconstructive evolution will extend the problem-solving power of genetic programming is still speculative. However, the hypothesis has been refined, the means for testing it have been simplified, the principles that underlie it have been better articulated, and the prospects for analysis of incremental results have been improved. We have shown (again) that mechanisms of adaptive variation can evolve as components of evolving problem-solving systems, and we have described reasons to believe that the best problem-solving systems of the future will make use of some such techniques. Only further experimentation will determine whether and when autoconstructive evolution will become the most appropriate technique for solving difficult problems of practical significance.

Acknowledgments

Kyle Harrington, Paul Sawaya, Thomas Helmuth, Brian Martin, Scott Niekum and Rebecca Neimark contributed to conversations in which some of the ideas used in this work were refined. Thanks also to the GPTP reviewers, to William Josiah Erikson for superb technical support, and to Hampshire College for support for the Hampshire College Institute for Computational Intelligence.

References

Angeline, Peter J. (1995). Adaptive and self-adaptive evolutionary computations. In Palaniswami, Marimuthu and Attikiouzel, Yianni, editors, *Computational Intelligence: A Dynamic Systems Perspective*, pages 152–163. IEEE Press.

Angeline, Peter J. (1996). Two self-adaptive crossover operators for genetic programming. In Angeline, Peter J. and Kinnear, Jr., K. E., editors, *Advances in Genetic Programming 2*, chapter 5, pages 89–110. MIT Press, Cambridge, MA, USA.

Barraclough, Timothy G., Birky, C. William Jr., and Burt, Austin (2003). Diversification in sexual and asexual organisms. *Evolution*, 57:2166–2172.

Beyer, Hans-Georg and Meyer-Nieberg, Silja (2006). Self-adaptation of evolution strategies under noisy fitness evaluations. *Genetic Programming and Evolvable Machines*, 7(4):295–328.

Diosan, Laura and Oltean, Mihai (2009). Evolutionary design of evolutionary algorithms. *Genetic Programming and Evolvable Machines*, 10(3):263–306.

Edmonds, Bruce (2001). Meta-genetic programming: Co-evolving the operators of variation. *Elektrik*, 9(1):13–29. Turkish Journal Electrical Engineering and Computer Sciences.

Eiben, Agoston Endre, Hinterding, Robert, and Michalewicz, Zbigniew (1999). Parameter control in evolutionary algorithms. *IEEE Transations on Evolutionary Computation*, 3(2):124–141.

Fodor, Jerry and Piattelli-Palmarini, Massimo (2010). *What Darwin got wrong*. New York: Farrar, Straus and Giroux.

Fontana, Walter (1992). Algorithmic chemistry. In Langton, C. G., Taylor, C., Farmer, J. D., and Rasmussen, S., editors, *Artificial Life II*, pages 159–210. Addison-Wesley.

Fry, Rodney, Smith, Stephen L., and Tyrrell, Andy M. (2005). A self-adaptive mate selection model for genetic programming. In *Proceedings of the 2005 IEEE Congress on Evolutionary Computation*, volume 3, pages 2707–2714, Edinburgh, UK. IEEE Press.

Gerhart, John and Kirschner, Marc (2007). The theory of facilitated variation. *Proceedings of the National Academy of Sciences*, 104:8582–8589.

Kantschik, Wolfgang, Dittrich, Peter, Brameier, Markus, and Banzhaf, Wolfgang (1999). Meta-evolution in graph GP. In *Genetic Programming, Proceedings of EuroGP'99*, volume 1598 of *LNCS*, pages 15–28, Goteborg, Sweden. Springer-Verlag.

Koza, John R. (1994). Spontaneous emergence of self-replicating and evolutionarily self-improving computer programs. In Langton, Christopher G., editor, *Artificial Life III*, volume XVII of *SFI Studies in the Sciences of Complexity*, pages 225–262. Addison-Wesley, Santa Fe, New Mexico, USA.

Langdon, William B. and Poli, Riccardo (2006). On turing complete T7 and MISC F–4 program fitness landscapes. In Arnold, Dirk V., Jansen, Thomas, Vose, Michael D., and Rowe, Jonathan E., editors, *Theory of Evolutionary Algorithms*, Dagstuhl, Germany. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany.

MacCallum, Robert M. (2003). Introducing a perl genetic programming system: and can meta-evolution solve the bloat problem? In *Genetic Programming, Proceedings of EuroGP'2003*, volume 2610 of *LNCS*, pages 364–373, Essex. Springer-Verlag.

Maynard Smith, John and Szathmáry, Eörs (1999). *The origins of life*. Oxford: Oxford University Press.

Nordin, Peter and Banzhaf, Wolfgang (1995). Evolving turing-complete programs for a register machine with self-modifying code. In *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 318–325, Pittsburgh, PA, USA. Morgan Kaufmann.

Ofria, Charles and Wilke, Claus O. (2004). Avida: A software platform for research in computational evolutionary biology. *Artificial Life*, 10(2):191–229.

Poli, Riccardo, Langdon, William B., and McPhee, Nicholas Freitag (2008). *A field guide to genetic programming*. `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`. (With contributions by J. R. Koza).

Ray, Thomas S. (1991). Is it alive or is it GA. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 527–534, University of California - San Diego, La Jolla, CA, USA. Morgan Kaufmann.

Schmidhuber, Jurgen (1987). Evolutionary principles in self-referential learning. on learning now to learn: The meta-meta-meta...-hook. Diploma thesis, Technische Universitat Munchen, Germany.

Schmidhuber, Jurgen (2006). Gödel machines: Fully self-referential optimal universal self-improvers. In Goertzel, B. and Pennachin, C., editors, *Artificial General Intelligence*, pages 119–226. Springer.

Silva, Sara and Dignum, Stephen (2009). Extending operator equalisation: Fitness based self adaptive length distribution for bloat free GP. In *Proceedings of the 12th European Conference on Genetic Programming, EuroGP 2009*, volume 5481 of *LNCS*, pages 159–170, Tuebingen. Springer.

Sipper, Moshe and Reggia, James A. (2001). Go forth and replicate. *Scientific American*, 265(2):27–35.

Smits, Guido F., Vladislavleva, Ekaterina, and Kotanchek, Mark E. (2010). Scalable symbolic regression by continuous evolution with very small populations. In Riolo, Rick L., McConaghy, Trent, and Vladislavleva, Ekaterina, editors, *Genetic Programming Theory and Practice VIII*. Springer.

Spears, William M. (1995). Adapting crossover in evolutionary algorithms. In *Proceedings of the Fourth Annual Conference on Evolutionary Programming*, pages 367–384. MIT Press.

Spector, Lee (2001). Autoconstructive evolution: Push, pushGP, and pushpop. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 137–146, San Francisco, California, USA. Morgan Kaufmann.

Spector, Lee (2002). Adaptive populations of endogenously diversifying pushpop organisms are reliably diverse. In *Proceedings of Artificial Life VIII, the 8th International Conference on the Simulation and Synthesis of Living Systems*, pages 142–145, University of New South Wales, Sydney, NSW, Australia. The MIT Press.

Spector, Lee, Klein, Jon, and Keijzer, Maarten (2005). The push3 execution stack and the evolution of control. In *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, volume 2, pages 1689–1696, Washington DC, USA. ACM Press.

Spector, Lee and Robinson, Alan (2002a). Genetic programming and autoconstructive evolution with the push programming language. *Genetic Programming and Evolvable Machines*, 3(1):7–40.

Spector, Lee and Robinson, Alan (2002b). Multi-type, self-adaptive genetic programming as an agent creation tool. In *GECCO 2002: Proceedings of the Bird of a Feather Workshops, Genetic and Evolutionary Computation Conference*, pages 73–80, New York. AAAI.

Suzuki, Hideaki (2004). *Design Optimization of Artificial Evolutionary Systems*. Doctor of informatics, Graduate School of Informatics, Kyoto University, Japan.

Tavares, Jorge, Machado, Penousal, Cardoso, Amilcar, Pereira, Francisco B., and Costa, Ernesto (2004). On the evolution of evolutionary algorithms. In *Genetic Programming 7th European Conference, EuroGP 2004, Proceedings*, volume 3003 of *LNCS*, pages 389–398, Coimbra, Portugal. Springer-Verlag.

Taylor, Timothy John (1999). *From Artificial Evolution to Artificial Life*. PhD thesis, Division of Informatics, University of Edinburgh, UK.

Teller, Astro (1994). Turing completeness in the language of genetic programming with indexed memory. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 136–141, Orlando, Florida, USA. IEEE Press.

Vafaee, Fatemeh, Xiao, Weimin, Nelson, Peter C., and Zhou, Chi (2008). Adaptively evolving probabilities of genetic operators. In *Seventh International Conference on Machine Learning and Applications, ICMLA '08*, pages 292–299, La Jolla, San Diego, USA. IEEE.

Woodward, John (2003). Evolving turing complete representations. In *Proceedings of the 2003 Congress on Evolutionary Computation*, pages 830–837, Canberra. IEEE Press.

Yabuki, Taro and Iba, Hitoshi (2004). Genetic programming using a Turing complete representation: recurrent network consisting of trees. In de Castro, Leandro N. and Von Zuben, Fernando J., editors, *Recent Developments in Biologically Inspired Computing*, chapter 4, pages 61–81. Idea Group Publishing.