Ontogenetic Programming

Lee Spector* †
lspector@hampshire.edu

Kilian Stoffel †
stoffel@cs.umd.edu

*School of Cognitive Science and Cultural Studies
Hampshire College
Amherst, MA 01002

†Department of Computer Science
University of Maryland
College Park, MD 20742

ABSTRACT

In nature, the structure and behavior of a mature organism is determined not only by its genetic endowment, but also by complex developmental processes that the organism undergoes while immersed in its environment (ontogeny). By contrast, an individual in a genetic programming system is generally expected to solve problems immediately, without the benefit of a developmental phase. Various morphological systems have been used in previous genetic programming systems to allow programs to "grow" into more complex forms prior to evaluation. Runtime memory mechanisms allow evolved programs to acquire information from their environments while they solve problems, and to change their future behavior on the basis of such information. *Ontogenetic programming* **combines these ideas to allow for runtime modification of program structure. In particular, an ontogenetic programming system includes program self-modification functions in the genetic programming function set, thereby allowing evolved programs to modify themselves during the course of a run. We demonstrate the ontogenetic programming methodology with the HiGP genetic programming system, and we show how ontogenetic programming can be used to solve problems that would not otherwise be solvable. We also discuss variants of the technique that could be used in other genetic programming systems.**

1 Phylogeny and Ontogeny

Biologists use the term *phylogeny* to refer to the developmental progression of a population or race through evolutionary time. Genetic programming systems use mechanisms inspired by those underlying biological phylogeny—in particular, genetic replication, mutation, and recombination combined with natural selection—to evolve programs that solve problems.

All but the simplest biological organisms also develop *as individuals*, both structurally and behaviorally, throughout their lives. Biologists refer to the developmental progression of an individual through its life span as *ontogeny*. In this paper we describe how rich ontogenetic components can be added to genetic programming systems, and we show how this can allow genetic programming to produce programs that solve more difficult problems.

Two sorts of ontogeny are evident in previously described genetic programming systems. The first consists of *morphological* components that allow programs to "grow" into different, perhaps more complex forms prior to execution. For example, in Gruau's technique of *cellular encoding* evolved programs are executed to produce neural networks; the networks that result from this developmental process are then assessed for fitness [Gruau 1994]. Similarly, Zomorodian has described a system in which evolved programs are executed to produce deterministic push-down automata [Zomorodian 1995]. Spector's automatically defined macros (ADMs) provide a morphological mechanism within the S-expression-based representational framework of traditional genetic programming; a program that includes ADMs is *macroexpanded* prior to execution, and the program that is executed may thereby be considerably larger and more complex than the evolved genotype [Spector 1995, 1996]. Morphological processes have also been used within other evolutionary computation paradigms; for example, Sims describes a system in which the morphological and behavioral components of virtual creatures are both described as directed graphs that evolve through the use of a graph-based genetic algorithm [Sims 1994]. Formal definitions of morphological processes in evolutionary computation, along with a good survey of past work, can be found in [Angeline 1995].

These sorts of morphological mechanisms may indeed add to the power of genetic programming, but they are simplifications of biological ontogeny in two important ways. First, they operate only prior to runtime; in contrast, biological ontogeny continues throughout the life span of an individual.

Second, they operate without environmental input; in contrast, a biological organism's environment may guide development in significant ways.

The second sort of ontogeny evident in previously described genetic programming systems involves the use of runtime memory mechanisms. Indexed memory [Teller 1994] and memory terminals [Iba et al. 1995] allow evolved programs to acquire and store information from their environments at runtime, and to use this information to guide future behavior. A program may thereby develop in certain ways throughout its "life span."

In theory, the ontogenetic mechanism provided by indexed memory is sufficient to support any developmental pathway for computer programs. One can imagine producing a program that includes a universal Turing machine; the program could periodically produce a new "more-developed" program in indexed memory and then begin to execute the new stored program. In practice, however, we cannot expect such remarkable programs to arise.

We use the term "ontogenetic programming" to describe an approach that combines the benefits of runtime memory with those of morphology. In ontogenetic programming, program self-modification operators are included in the set of functions that may be used by evolved programs. By use of these operators, an evolved program can dynamically change its own structure, and thereby change its future behavior, during the course of a run. A program's self-modification strategy is itself evolved; it may be arbitrarily complex and it may be conditionalized on runtime environmental inputs.

In our current implementation the self-modification and problem-solving components of programs are intermingled within a single program structure. But this is not essential; in other variants of the technique one might keep these components segregated. In either case, evolved programs can influence their own future behavior via direct manipulation of their own program structure.

Relations between evolution and ontogeny in evolutionary computation have previously been studied with genetic algorithms that produce neural networks (for example, [Parisi et al. 1992; Batali 1994]). Other work has explored the incorporation of runtime morphological development mechanisms into evolutionary computation frameworks (for example, [Dallaert and Beer 1994; Hemmi et al. 1994]). The work described in this paper applies related ideas to the production of computer programs that develop over time.

In the following sections we first discuss the adaptive utility of ontogeny. We then demonstrate ontogenetic programming using the HiGP genetic programming system. HiGP operates on linear programs for a stack-based virtual machine; program-modification mechanisms can be particularly simple when applied to these sorts of linear programs. The technique is not limited to such systems, however; at the end of the paper we discuss ontogenetic programming with more traditional S-expression-based representations.

2 Why Develop?

To some extent biological ontogeny is mandated by the physics of biological reproductive mechanisms. These produce, initially, very small structures; in many niches large size is adaptive (because, for example, it allows for faster movement or more effective hunting), and developmental processes are necessary to produce large structures. Ontogeny is therefore adaptive, if only because large size is adaptive, and because genes can produce large structures only by orchestrating developmental processes.

The situation is rather different for computer programs in a genetic programming system, at least with respect to size. Newly created programs may be of any size whatever, and it is not clear that large size is itself adaptive in any interesting niches. But developmental processes may nonetheless be adaptive for computer programs—developmental control of *size* may or may not be particularly useful, but developmental control of other phenotypic features is clearly useful in certain domains. More generally, developmental control of an arbitrary feature *f* of an individual is likely to be useful in two cases: 1) when there is an adaptive advantage to having an *f* value at maturity that differs significantly from the *f* values of new individuals, or 2) when there is an adaptive advantage to having different *f* values at different stages of an individual's life.

The second case applies to features of computer programs for many domains. Consider a program for an agent in an adventure game. It might be advantageous for such a program to change several times throughout the course of a run; it might be useful to have an initial, high-risk, exploratory phase, a more focused goal-seeking phase, and an analytical endgame phase. Each phase might be best served by rather different programs, and a program that could develop through several different forms during the course of its life span might have a significant advantage. This sort of development is enabled in ontogenetic programming by building the capability for self modification into the programs themselves—this allows them to evolve strategies for changing themselves as they run.

Consider the more abstract *sequential regression* problem, which shares features with many sequence prediction problems. As in ordinary symbolic regression [Koza 1992], the goal in sequential regression is to produce a program that returns the appropriate *y* value for each *x* value in a data set. Often the data set is generated by a known target function, and the regression problem can be viewed as one of rediscovering the target function from the data. In the sequential regression problem we stipulate also that the *x* values will be presented in a particular order. Programs have a "life span" insofar as they are run multiple times, one for each *x* value in the fitness-testing range, and insofar as they always encounter these *x* values in the same order. Development may be useful for sequential regression problems because different programs may be most appropriate for different ranges of the target function. It may also be useful because transitions from one value to the next may be better mediated by developmental operators than by domain operators.

Consider the particular sequential regression problem of rediscovering the target function x^2 over the non-negative integers. Suppose that we are using a function set that includes only addition and subtraction operators. Using ordinary genetic programming it is impossible to produce a correct program, because the provided function set is only capable of generating linear functions. But in an ontogenetic programming system with sufficiently powerful self-modification operators, a program can modify itself during the $x = n$ run to allow the subsequent run to produce a correct result for $x = n + 1$. We have in fact witnessed the evolution of such an ontogenetic program; upon each execution it inserts an additional "+ *x*" sequence into itself, thereby preparing itself for the next value. Although evolved with fitness cases that ranged only from 1 to 19, the program is general and works correctly for ranges from 1 to any positive integer (given unbounded space for program growth). In this case the ontogeny of the individual is properly matched to the dynamics of the individual's life course; as a result, the developmental process is able to extend the power of the available function set.

In some cases the benefits of development can be obtained more simply through the use of conditionals. For example, a program that develops in such a way that it replaces a body of code *a* with a body of code *b* at time *t* could in some environments be implemented with a conditional that executes either *a* or *b* depending on the result of a comparison of the current time to *t*.[1] Development is nonetheless more powerful; for example, a developing program may develop through an unbounded number of forms throughout its life span, while the number of execution paths for any conditionalized program is fixed.

Summarizing, there are a variety of situations in which development may be useful, for a variety of different reasons. Any problem for which programs have "life spans," in the sense used above, is a candidate for the application of ontogenetic programming. For such problems ontogeny may provide a capability that would otherwise be beyond the reach of the available function set.

In the next section we describe the HiGP implementation of ontogenetic programming, and in the subsequent section we demonstrate the utility of the approach for a binary sequence prediction problem.

3 Ontogenetic HiGP

HiGP is a new high-performance genetic programming system that combines techniques from string-based genetic algorithms, S-expression-based genetic programming systems, and high-performance parallel computing [Stoffel and Spector 1996]. The result is a fast, flexible, and easily portable genetic programming engine with a clear and efficient parallel implementation. The parallel version of HiGP scales nearly linearly with the number of available processors. HiGP manipulates and produces linear programs for a stack-based virtual machine (as in [Perkis 1994]), rather than the tree-structured S-expressions used in traditional genetic programming.

HiGP programs are executed on a virtual machine that is similar to a pushdown automaton. The virtual machine consists of three components: an input tape containing a linear program, a pushdown stack, and a finite-state control unit. The contents of the input tape are restricted to a small set of words that have been defined as HiGP operators. The contents of pushdown stack are restricted to double precision floating point numbers. The finite-state control unit reads the input tape and executes, for each word, the function call for the corresponding operator. The operators may perform arbitrary computations and manipulate the values on the stack. They may also reposition the read head on the input tape; this allows for the implementation of conditionals and loop structures. Return values are generally read from the top of the stack at the end of program execution.

The system includes two basic stack operators, `pop` and `dup`. The `pop` operator removes the topmost element from the stack, while the `dup` operator pushes a duplicate of the top element onto the stack. The system also includes a family of `push` operators that correspond to the terminal set in a traditional genetic programming system; each `push` operator pushes a single pre-determined value onto the stack.

The system also includes a `noop` operator that does nothing. This is necessary because all programs in the system have the same length, and because we do not wish to predetermine the number of actual problem-solving operators that should appear in solution programs. With the inclusion of the `noop` operator the fixed program size becomes a size *limit*, analogous to the depth limits used in S-expression-based genetic programming systems. "Shorter" programs are encoded by filling in extra program steps with `noop`s.

Any additional, problem-specific operators must take their input values from the stack and must push their results back onto the stack. When there are not enough values on the stack for a problem-specific operator it is skipped by the finite-control unit and the stack remains untouched (as in [Perkis 1994]). Additional information about HiGP may be found in [Stoffel and Spector 1996].

The ontogenetic version of HiGP results from including the following program self-modification operators in the function set:

segment-copy copies a part of the linear program over another part of the program. The function takes three arguments from the stack: the start position of the segment to copy, the length of the segment, and the position to which

[1] We thank an anonymous reviewer for drawing this possibility to our attention.

the segment should be copied. All positions are calculated relative to the current instruction. If there are not three values on the stack the instruction is skipped.

shift-left rotates the program to the left. The call takes one argument from the stack: the distance by which the program is to be rotated. If there is no value on the stack the program is rotated by one instruction to the left.

shift-right rotates the program to the right. The call takes one argument from the stack: the distance by which the program is to be rotated. If there is no value on the stack the program is rotated by one instruction to the right.

The position of the current instruction pointer is not changed by the execution of an ontogenetic operator. For example, if instruction #23 is a `segment-copy`, then after its execution instruction #24 will be executed, regardless of the fact that the *old* instruction #23 may now have been moved elsewhere.

4 Example: Binary Sequence Prediction

In this section we show how the ontogenetic version of HiGP can solve a problem that cannot be solved by the ordinary version of HiGP. We also show that the addition of indexed memory to the ordinary version of HiGP is not sufficient to produce solutions to this problem; the ontogenetic extensions provide benefits that indexed memory does not.

Consider a binary version of the sequential regression problem described above, with a target function that repeats the values [0 1 0 0 0 1] as *x* increases. The initial portion of the target function is {(0, 0) (1, 1) (2, 0) (3, 0) (4, 0) (5, 1) (6, 0) (7, 1) (8, 0) (9, 0) (10, 0) (11, 1) (12, 0)...}. The goal is to produce a program that returns the appropriate *y* value for each *x* value; the program will be run once for each *x* value, and the *x* values will be presented in order from 0 to some number *n*. In particular, we assessed fitness by testing each program on the range [0–19]. We use a "wrapper" function [Koza 1992] that maps positive *y* values to 1, and negative *y* values to 0. In other words, it is not necessary for an evolved program to produce the exact values 0 and 1; it is sufficient for it to produce either 0 or any negative number in place of each 0, and any positive number in place of each 1. This problem can be thought of as a binary sequence prediction problem, related to many other sequence prediction problems in the literature. In particular, Iba et al. describe a related binary oscillation task [Iba et al. 1995].

The function set for this problem consists of the 2-argument addition function `+`, the 2-argument subtraction function `-`, the 2-argument multiplication function `*`, the 2-argument protected division function `%` [Koza 1992], and three `push` functions: `push-x` (for the independent variable *x*), `push-0`, and `push-1`. The HiGP functions `dup` and `noop` were also included. Although this function set is simple it is nonetheless general and powerful; for example, one can obtain the effects of many conditionals through the clever use of arithmetic. For each fitness case the stack was initialized with a single copy of *x*, and the result was read from the top of the stack at the end of program execution.

We conducted 100 runs of the ordinary (non-ontogenetic) version of HiGP on this problem. We used a population size of 100, a maximum program size of 30, a crossover rate of 90%, and a reproduction rate of 10%. We allowed each run to continue for up to 20 generations. No correct solutions were produced by any of the 100 runs. We conclude that the problem cannot reliably be solved by ordinary HiGP and the given parameters.

We then conducted 100 runs of the ontogenetic version of HiGP on this problem (adding the `segment-copy`, `shift-right` and `shift-left` functions to the function set). We used the same population size and other parameters as in the non-ontogenetic case. 12 completely correct solutions were produced by the 100 runs. 10 of these solutions were general; although fitness was assessed only over the range [0–19], these programs produce correct results over the range [0–39], and they appear to be correct to any limit. The following is one of the correct evolved programs:

```
push-x % - shift-left push-x noop * *
dup % - + push-x % dup % shift-right
dup shift-left push-x shift-right * %
shift-right * + shift-right - - push-x
```

Because indexed memory also provides a limited ontogenetic capability, we also conducted 100 runs of the HiGP with indexed memory on this problem. We added a 30-element indexed memory, a 2-argument `write` function that stores the value of one argument in the location indexed by the other, and a 1-argument `read` function that pushes the indexed element of the memory onto the stack. We used the same population size and other parameters that were used for the runs described above. No correct solutions were produced by any of the 100 runs. We conclude that indexed memory does not provide sufficient power for HiGP to reliably solve this problem with the given parameters.

In summary, ontogenetic programming allows for the production of binary sequence prediction programs that change themselves in appropriate ways during the course of their "life spans" through the *x* range. This allows ontogenetic programming to reliably produce correct solutions to the binary sequence prediction problem, whereas the non-ontogenetic and indexed memory versions of our system were unable to produce any correct solutions.

5 Ontogenetic Programming with S-Expressions

Although the program self-modification functions of ontogenetic programming are easiest produce and to explain for linear programs, variants of the technique can be applied to the S-expression-based program representations of traditional genetic programming [Koza 1992].

One way to apply ontogenetic programming to S-expression-based programs is to include a `subtree-copy` function in the function set. The `subtree-copy` function takes two integer arguments, each of which indexes a subtree of the program through some standard (e.g., depth-first) traversal. The effect of a call to the `subtree-copy` function is to replace the subtree rooted at one of the indices with a copy of the subtree rooted at the other.

We have experimented with versions of the `subtree-copy` function, and although we have had some successes (e.g., for the x^2 sequential regression problem described above), several problems remain. One problem is that it is awkward to modify an S-expression during its execution. We avoided this problem by applying the `subtree-copy` operations to a copy of the program, and by replacing the program with its modified copy at the end of each execution. This means that a program could develop *between* the successive executions of its "life span," but not *during* the executions. In contrast, our HiGP implementation allows for program development during each single execution. This limitation of the `subtree-copy` model could be removed by using a more flexible S-expression evaluator.

A second problem with the `subtree-copy` model of ontogenetic programming is that the index arguments refer to specific subtrees of the entire program, indexed relative to the root of the S-expression. This means that it is unlikely that a particular call to `subtree-copy` will be transportable from one program to another; it will have a completely different effect in each program context into which it is placed. Crossover will therefore rarely produce high-fitness children from high-fitness parents — the fitness landscape will have sharp discontinuities. One way to mitigate this problem is to use a 3-argument `structured-subtree-copy` control structure. As with `subtree-copy`, the first two arguments to `structured-subtree-copy` produce integer indices. But the third argument is a *result-producing branch* that is executed to produce a value from the call to `structured-subtree-copy`. As a side-effect of each call to `structured-subtree-copy`, a subtree-copy operation is performed *within the result-producing branch*. Since the indices refer only to subtrees within the result-producing branch, and since they are indexed relative to the root of the result-producing branch, it is reasonable to assume that the entire call to `structured-subtree-copy` will be transportable to other program contexts.

A third problem with the `subtree-copy` model of ontogenetic programming is that it may produce arbitrarily large programs during execution. Whereas the HiGP program-modification functions never change the size of a program, the replacement of a small subtree by a large subtree in an S-expression program will indeed produce a larger program. Several techniques may be used to limit this growth. These include absolute depth limits that are enforced by converting offending `subtree-copy` calls into `noop`s, and depth-based indexing schemes that ensure that replacement subtrees are selected only from those with acceptable sizes.

An alternative mechanism for ontogenetic programming with S-expression-based programs makes use of *dynamic ADFs* and *dynamic ADMs*. With ordinary automatically defined functions (ADFs [Koza 1994]) and automatically defined macros (ADMs [Spector 1995, 1996]) each individual has a fixed set of evolved sub-functions/macros that may be used by the main program and by each other. In contrast, dynamic ADFs and ADMs may be redefined by the individual at runtime, and the new definitions may be constructed on the basis of information in the individual's environment or memory. The implementation involves the inclusion function/macro definition and calling operators (e.g. variants of Lisp's DEFUN, FUNCALL, etc.) in the function set. One can refer to the dynamically defined functions/macros using integer labels; this allows one to use a mechanism much like indexed memory, with the memory containing callable and redefinable functions and macros rather than simple values. Special but straightforward measures must be taken to avoid problematic recursive definitions. Dynamic ADFs and ADMs allow programs to modify themselves at runtime through module redefinition, rather than through direct manipulation of program code. We are currently investigating the utility of this technique.

6 Conclusions

We have shown that it is possible to use genetic programming to produce programs that themselves develop in significant, structural ways over the course of a run. We use the term "ontogenetic programming" to describe our technique for achieving this effect, which involves the inclusion of program self-modification functions in the genetic programming function set. We described cases in which this may be useful, and we demonstrated the application of our HiGP implementation of ontogenetic programming to a binary sequence prediction problem. Although HiGP manipulates linearly-structured programs, and the program self-modification functions are particularly simple in this case, we also described several variants of the technique that are appropriate for more traditional S-expression-based program representations.

Acknowledgments

Mark Feinstein helped to develop our initial interest in ontogeny, and to refine our understanding of its role in biological systems. Discussions at the 1995 AAAI Fall Symposium on Genetic Programming [Siegel 1995] helped to further refine our approach to ontogenetic programming. We also acknowledge the helpful comments of several anonymous reviewers. This research was supported in part by grants from ONR (N00014-J-91-1451), AFOSR (F49620-93-1-0065), and ARPA contract DAST-95-C0037.

Bibliography

Angeline, P.J. 1995. Morphogenic Evolutionary Computations: Introduction, Issues, and Examples. In *Proceedings of the Fourth Annual Conference on Evolutionary Programming,* edited by J.R. McDonnell, R.G. Reynolds, and D.B. Fogel. Cambridge, MA: The MIT Press.

Batali, J. 1994. Innate Biases and Critical Periods: Combining Evolution and Learning in the Acquisition of Syntax. In *Artificial Life IV: Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems,* edited by R.A. Brooks and P. Maes. pp. 160–171. Cambridge, MA: The MIT Press.

Dellaert, F., and R.D. Beer. 1994. Toward an Evolvable Model of Development for Autonomous Agent Synthesis. In *Artificial Life IV: Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems,* edited by R.A. Brooks and P. Maes. pp. 246–257. Cambridge, MA: The MIT Press.

Gruau, F. 1994. Genetic Micro Programming of Neural Networks. In *Advances in Genetic Programming,* edited by Kenneth E. Kinnear, Jr. pp. 495–518. Cambridge, MA: The MIT Press.

Hemmi, H., J. Mizoguchi, and K. Shimohara. 1994. Development and Evolution of Hardware Behaviors. In *Artificial Life IV: Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems,* edited by R.A. Brooks and P. Maes. pp. 371–376. Cambridge, MA: The MIT Press.

Iba, H., T. Sato, and H. de Garis. 1995. Temporal Data Processing Using Genetic Programming. In *Proceedings of the 6th International Conference on Genetic Algorithms, ICGA-95,* edited by Larry J. Eshelman, pp. 279–286. San Fransisco: Morgan Kaufmann Publishers, Inc.

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

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

Parisi, D., S. Nolfi, and F. Cecconi. 1992. Learning, Behavior, and Evolution. In *Toward a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life,* edited by F.J. Varela and P. Bourgine. pp. 207–216. Cambridge, MA: The MIT Press.

Perkis, T. 1994. Stack-Based Genetic Programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*. pp. 148–153. IEEE Press.

Siegel, E.V., editor. 1995. *Collective Brainstorming at the AAAI Symposium on Genetic Programming,* http://www.cs.columbia.edu/ evs/gpsym95.html.

Sims, K. 1994. Evolving 3D Morphology and Behavior by Competition. In *Artificial Life IV: Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems,* edited by R.A. Brooks and P. Maes. pp. 28–39. Cambridge, MA: The MIT Press.

Spector, L. 1995. Evolving Control Structures with Automatically Defined Macros. In *Working Notes of the AAAI Fall Symposium on Genetic Programming*. pp. 99–105. AAAI Press.

Spector, L. 1996. Simultaneous Evolution of Programs and their Control Structures. In *Advances in Genetic Programming 2,* edited by Peter J. Angeline and Kenneth E. Kinnear, Jr. Cambridge, MA: The MIT Press. In press.

Stoffel, K., and L. Spector. 1996. High-Performance, Parallel, Stack-Based Genetic Programming. In *Proceedings of The Genetic Programming 1996 Conference.* Cambridge, MA: The MIT Press.

Teller, A. 1994. The Evolution of Mental Models. In *Advances in Genetic Programming,* edited by Kenneth E. Kinnear, Jr. pp. 199–219. Cambridge, MA: The MIT Press.

Zomorodian, A. 1995. Context-Free Language Induction by Evolution of Deterministic Push-Down Automata Using Genetic Programming. In *Working Notes of the AAAI Fall Symposium on Genetic Programming*. pp. 127–133. AAAI Press.