

A Historical Perspective on the Evolution of Executable Structures

Peter J. Angeline
Natural Selection, Inc.
angeline@natural-selection.com

Abstract

Genetic programming (Koza 1992) is a method of inducing behaviors represented as executable programs. The generality of the approach has spawned a proliferation of work in the evolution of executable structures that is unmatched in the history of the subject. This paper describes the standard approach to genetic programming, as defined in Koza (1992), and then presents the significant studies that preceded its inception as well as the diversification of techniques evolving executable structures that is currently underway in the field.

1 Introduction

One of the ultimate goals of machine learning is to create an autonomous intelligence that can adapt its behavior appropriately for any situation it encounters. The manipulation of potential behaviors is necessarily dependent on the method of representation for the range of possible behaviors. A poor representation limits what behaviors can be created while a good representation permits the induction of any potentially useful sequence of actions.

An obvious candidate representation for behaviors of an autonomous intelligent machine is a standard programming language. Programming languages have the advantage of being Turing complete, meaning they can represent any computable behavioral sequence. But the syntactic “sugar” of most high-level programming languages pose a difficult problem for any type of intelligent search method, especially naive approaches to inducing programs. A single missing or misplaced syntactic marker can render an otherwise perfectly correct program unexecutable and hence useless. This situation becomes aggravated when an evolutionary process is used to induce appropriate programs. Applying naive stochastic operations to a syntax-heavy representation for programs typically produces only unexecutable offspring.

Genetic programming represents one approach to the induction of appropriate behaviors using an evolutionary computation. Its definition and demonstration on a variety of engineering problems in Koza (1992) represents the basis of much of the current work in this area. The general insight of genetic programming is that by distilling a programming language down to its behavioral core, programs become more evolutionarily manipulatable and hence the space of possible behaviors can be searched more effectively. Koza (1992) also demonstrates the versatility of this method.

This paper aims to describe the variety of methods investigated for the evolution of executable structures. The following sections review the progression from the earliest computational attempts at program evolution through the diversity of methods that define the current state-of-

the-art. A companion article tracing additional aspects of current trends in the evolution of executable structures in general and genetic programming in particular can be found in Angeline (1996).

1 Standard Genetic Programming

The publication of Koza (1992) was a significant event in the history of evolutionary program induction. The following description of the technique defined in Koza (1992) provides a historical touch stone that places the significance of associated research that preceded and followed it into perspective.

The initial step for creating a standard genetic program is to define the set of primitive functions and terminals that will comprise the evolving programs. This set of primitives defines the verbs and nouns of the “programming language” for the problem and is most often defined using high-level concepts that are computationally germane to the task. Each function included in the function set must take a specified number of input parameters, while terminals, by definition, take no parameters.

The basic representational structure of a genetic program is a directed, acyclic graph, also known as a *tree*, with symbols from the defined language associated with each position in the tree. Accordingly, terminals occupy the leaves of the trees while functions the internal positions. The overall structure of a program tree is determined by the number of input parameters to the functions that occupy the tree’s internal points. A function that takes three arguments, for instance, will have three associated subtrees below it, one per input argument. The number of parameters required for a function is the only syntactic constraint associated with genetic programming trees. The tree structure significantly reduces the syntax of programs and permits the definition of manipulation routines that generate only viable programs during the search.

Each terminal, function, and function parameter has an associated *type* in the traditional sense. The return type of a terminal is the type of data value it returns, for instance a floating point value. In general, a single return type is assumed for all terminals, functions, and function parameters. This is known as the *closure principle* (Koza 1992). Consequently, the return value of any terminal or function can be used as the input value to any parameter of any function. While this may be an unrealistic model of a program from a computer scientist’s point of view, it is a prudent choice from an evolutionary perspective since it removes all type consideration from the syntax being evolved

As described in Koza (1992), genetic programming is a genetic model of evolution and as in genetic algorithms (Holland 1975) its chief mechanism for creating unique offspring is crossover. The form of crossover used in genetic programming is a natural consequence of the tree representation, and is fittingly named *subtree crossover*. Subtree crossover proceeds as follows. Given two parents, a randomly selected subtree is identified in each and swapped between them. This simple operation preserves all the associated constraints of the tree representation given that the parents are syntactically correct. Koza (1992) biases subtree selection away from the leaves of the parent trees using a *leaf-frequency parameter* which defines the relative probability of selecting a leaf node or internal node of a parent tree when performing subtree crossover. The value for this

parameter in Koza (1992) is 10%, meaning leaves are selected about 10% of the time during crossover.

As in genetic algorithms, genetic programming treats mutation as a background operator. In fact, it is not uncommon for a standard genetic program to use no mutation operation at all. Current trends in the evolutionary computations suggest a new found appreciation of mutation's role, as described below.

While it is claimed in Koza (1992) that genetic programs place no restrictions on the size and shape of the evolving programs, this is not strictly true. If no limits were set for the evolving programs, they would grow without bound and very quickly saturate the available computational resources. As a rule, all genetic programming implementations provide parameters that enforce hard limits on program size, either by specifying a maximal depth or a maximum number of nodes for a generated program tree.

Depth limitation, used extensively in Koza (1992), sets a maximal depth for all genetic programs in the population. This parameter directly limits both the allowable size for an evolving program and the maximum execution time available. Node limitation places a maximum limit on the number of nodes allowed in a tree. This also places an upper limit on size and execution time but may be more flexible than depth limitation since it avoids introducing an arbitrary constraint on the depth of the program. If the goal is to evolve structures with only a minimal specification of its characteristics, a limitation based on a maximum number of nodes seems more consistent with this goal. In any case, as noted in Koza (1992), any reasonable limitation on program size is not much of a limitation since the number of programs in the search space will still be very large.

In order to evolve a program, its ability to implement a desirable solution must be measured. In standard genetic programming, a fitness function typically evaluates a program by executing it within the intended environment and rating the resulting behavior. This is strictly counter to the philosophy of fitness evaluation in genetic algorithms where an individual's structure is often graded directly. A genetic program must therefore make structural alterations based on feedback on the program's behavior when executed. Such an induction is in general non-trivial since the association between function and structure is rarely a unique mapping.

1 The Pre-History of Evolution and Executable Structures

Alan Turing was possibly the first to experiment with a form of evolution for the purposes of learning behaviors. In his classic paper introducing the Turing test (Turing 1950), Turing speculated on the possibility of using an automated evolutionary process to "educate" a machine until it could achieve sufficient proficiency to challenge a human tester. Turing felt such an evolutionary process would ultimately be too slow and could be improved through the interaction of a human instructor. Turing admits "I have done some experiments with one such child-machine, and succeeded in teaching it a few things, but the teaching method was too unorthodox for the experiment to be considered really successful" (Turing 1950, pp. 457).

Friedberg (1958) and Friedberg, Dunham, and North (1959) present a significant early attempt to induce programs using an evolution inspired procedure. Friedberg (1958) used a form of credit

assignment that rewarded individual instructions based on the success of the whole program. Random instructions were periodically used to replace those not contributing to successful results. This form of credit assignment was replaced by a more guided, progressive learning method in Friedberg, Dunham, and North (1959) that ultimately improved overall performance to a level competitive with other generate and test methods.

Fogel, Owens and Walsh (1966) is another historical landmark in the evolution of programs. In this book, which defined the field of evolutionary programming, finite state machines were evolved in order to induce predictive behaviors. The evolutionary approach was to create a specific form of mutation for each unique component of a finite state machine. No form of crossover was required. Multiple structural mutations were applied to a parent so that the created child contained some structural differences while largely remaining similar to the parent. The ability of a candidate machine to produce the correct predictive behavior was scored with those machines that scored best being used as parents for the next generation. Fogel, Owens and Walsh (1966) argued that this technique was an important complement to the human centered methods of early artificial intelligence. A similar approach was also used in a more recent incarnation of evolutionary programming geared towards evolutionary function optimization (Fogel 1995).

Many of the important features of genetic programming are contained within the evolutionary programming technique of Fogel, Owens and Walsh (1966) offered more than 25 years earlier than Koza (1992). The simplifying definition of a program as a structure with minimal syntax is clearly present in the form of the finite state machine while the structure-specific mutations preserve the minimal syntactic requirements. A third similarity is the fact that both early evolutionary programming and genetic programming evaluate the evolved structure's behavior, rather than its structure directly, as in genetic algorithms. Unlike the structural mutation approach of evolutionary programming, genetic programming routinely makes significant structural alterations when producing offspring with subtree crossover. Genetic programming then represents an interesting hybrid between genetic algorithms and evolutionary programming.

The first usage of subtree crossover to evolve symbolic expressions is due to Cramer (1985) which tersely describes experiments involving a tree-based language referred to as TB. The task was to evolve a simple two input, single-output multiplication function using in what was little more than an assembly language. TB and subtree crossover were developed after less successful attempts at evolving simple string-based programs, also described in the paper. Smith (1980; 1983) is identified in the paper as a source of inspiration for the subtree crossover operation, specifically his limitation of where crossover can be applied in an evolving structure. Cramer (1985) also defines a version of subtree inversion but it is unclear if this was used in the described experiments.

About the same time as Cramer (1985), other researchers were attempting to evolve limited symbolic organizations similar in spirit to work in Smith (1980; 1983) on variable length conditions in classifier systems. Hicklin (1986) evolved LISP expressions in his experiments on evolving game playing programs. Offspring were created through the combination of two parents using a unique recombination method. Beginning at the roots, Hicklin (1986) copied the common elements of the parents into the child. When the parents did not agree on an element, the remainder of the subtree

of one parent, chosen at random, was copied into the child. Consequently, a child was an amalgam of several subtrees from both parents. In addition, a form of point mutation was applied to reintroduce lost information back into the population.

Following Hicklin (1986), Fujiki (1986) and later Fujiki and Dickson (1987) described experiments evolving a single LISP conditional expression for game playing with operators that more closely followed genetic algorithms. Crossover here entailed recombining the condition-action pairs between two individuals using a form of single-point crossover. Likewise, an inversion operator reordered condition-action pairs while mutation was used to introduce novel condition-action pairs into the population.

Bickle and Bickle (1987) evolved rule sets where each rule was a task specific symbolic expression represented as a tree. Tree-based versions of mutation and inversion were used to modify individual rules while a variant of string-based crossover recombined the ordered list of rules between parents.

Dickmanns, Schmidhuber and Winklhofer (1986) investigated the evolution of simple PROLOG programs represented as variable length strings which they called “plans”. Also following genetic algorithms, they defined suitable operations for crossover, inversion, mutation, and element deletions and insertion. They demonstrate the ability of their approach by constructing a simple robot controller and an integer sorting algorithm.

The application of subtree crossover to general tree-based symbolic expressions was reconsidered in Koza (1989) which investigated an early form of genetic programming, referred to as hierarchical genetic algorithms. Demonstrations of the technique included planning and symbolic regression. Koza (1989) recognized the importance of minimizing syntactic issues when evolving programs and explicitly recommended the use of a single return type for each function and terminal defined for the task-specific language, a point that was implicitly observed in much of the preceding work.

1 The Current State of Evolutionary Program Induction

With the advent of Koza (1992), researchers in genetic programming began to explore the potential of the paradigm. This rapid diversification is due as much to the breadth of topics and insights offered in Koza (1992) as the malleability of the method and the generality of genetic programming’s inherent concept of program.

Extensions to the basic genetic programming paradigm can be separated into two fundamental research threads. The first attempts to remove the restrictions on the form of the evolved programs by introducing specific constructs typically found in traditional programming languages, such as modularity, state variables, and multiple data types, without compromising evolvability. The second concentrates on the understanding, improvement and extension of the evolutionary process responsible for inducing the programs.

1.1 Genetic Programming and the Schema Theorem

Given genetic programming's obvious association with genetic algorithms, and the prominent role the schema theorem plays in that research, one research question asks if there is a similar theorem to be proved for the evolution of programs. The chief difficulty each study on this topic has encountered has been how to define the concept of a schema for variable length parse trees. Recall Holland (1975) defines a schema as an equivalence class of bit strings using the symbols $\{0,1,\#\}$ where “#” denotes a “don't care” symbol.

Koza (1992), attempting a literal translation of Holland's concept, defines schema for genetic programs directly as a set of trees all having one or more subtrees in common, but does not carry this definition any further than its statement. O'Reilly and Oppacher (1995) devise two definitions for GP-schema, both of which also identify a schema as a collection of trees possessing a particular subtree. They go on to define a version of the schema theorem using their definitions and then acknowledge that it is based on untenable assumptions. Rosca and Ballard (1997) define a rooted tree schema as the set of trees sharing identical subtrees beginning at the root of the tree. They apply their definition to investigate the dynamics of program tree growth during evolution. While interesting, their method unevenly biases the defined set of equivalence classes to structures based at the root of the trees. This would be analogous to limiting a genetic algorithm's schema to only include schema where the first few bits can only contain literal symbols and not the “#” symbol.

The major obstacle to defining any suitable definition for schema for genetic programs is due to the decoupling of position with semantics in the evolving parse trees. The schema theorem is based on the assumption that the semantics of the representation are tied to the position of the bit in the string. This assumption permits the semantics to be reduced to the simple syntactic notation used by Holland (1975) to define the similarity sets known as schema. In genetic programming, the semantics of a subtree are not tied to its position in the tree, they are self-contained within the subtree itself. Consequently, any approach based on modeling syntax alone will not model the correct phenomena.

1.1 Evolving Modular and Recursive Symbolic Expressions

One of the most appealing aspects of conventional programming languages is modularity and reuse of subroutines. The importance of discovering appropriate modularizations for the automatic construction of intelligent behaviors has been known since the early days of artificial intelligence (Simon 1969). Reuse and modularization has played an important role in the extension of genetic programming as well.

Koza (1992; 1994) and Angeline and Pollack (1992;1994) independently describe early methods for incorporating modularity into genetic programs. Koza (1992; 1994) describes *automatically defined functions*, which allows the user to define the hierarchy of subroutines available to evolving programs. Koza's approach allows the language of each subroutine, including which subroutines are available to be called, to be hand tailored. Limited, hierarchical call sequences are permitted allowing the induction of only non-recursive programs. Koza (1994) describes a slightly more flexible form of this technique that dynamically alters some of the characteristics of the subroutines. Angeline and Pollack (1992; 1994) take a different approach. They extend the set of

mutations available to the genetic program to include mutations that create subroutines by dynamically extracting random subtrees from evolving programs. The extracted subtrees are used to define new functions that are added to the representational language of the genetic program. The extracted subtree is then replaced by a call to the newly created function. Hierarchical calling structures emerge when the random subtree selected contains a reference to a previously created subroutine. A reverse mutation that replaces a created function symbol with its subtree definition is included to introduce a non-linear sequence of subroutine definitions. Angeline and Pollack (1994) demonstrate that the dynamics of the genetic program are sufficient to implicitly determine which of the randomly selected subroutines are the most evolutionarily viable and automatically excise those that are unusable.

Rosca and Ballard (1996) take an approach similar to Angeline and Pollack (1994) but with a more explicit evaluation method. They begin with the hypothesis that the contribution of a particular subtree in a genetic program can be estimated directly. Given such an estimation, those subtrees that are most important to the solution can be used to define new subroutines. Hierarchical call sequences again emerge when previously defined subroutines are included in the definition of more recent subroutines.

There are two basic similarities among the above methods. First, they all induce non-recursive solutions. The basic difficulty with inducing recursive structures is the stopping criteria. If recursion is introduced naively into a genetic program, infinite recursive programs are usually the norm, ultimately defeating the evolutionary process. The second similarity is the form of modularity employed. All three assume the typical programmatic calling method, namely jumping to a subroutine and executing it anew on each encounter of its symbol in the program. However, there is no reason to believe that this particular form is more evolvable than others simply because it is favored by human programmers.

1.1 Evolving Programs with Multiple Data Types

In order to simplify the manipulation of evolving structures genetic programs typically assume a single return type for each language element. Relaxing this constraint is not difficult and can often significantly reduce the search space by eliminating a larger number of untenable or suboptimal solutions. Researchers employing multiple data types generally use a grammar to describe the legal constructions with the set of functions and terminals provided. When manipulating a program using crossover or mutation, the return type of the subtree to be added is enforced to be identical to that which was removed, thus preserving the restrictions described by the grammar.

Hicklin (1986) was the first to employ a grammar to restrict the possible constructions in the population, although exactly how the grammar is applied is not clear. The intention to use the grammar to allow this genetic algorithm to produce evolved programs with multiple data types is clearly evident. Formal experiments in Montana (1995) showed not only that multiple data types could be evolved but also that they could induce solutions to some problems more efficiently in some situations. Other work along these lines include Haynes (1996) and Gruau (1996).

An important outstanding question associated with this extension is its effect on the connectivity and evolvability of the search space. It seems plausible that should a sufficiently complex grammar

be used to limit feasible programs it may be difficult in some circumstances to create untested programs that conform to the grammar. In this case, the population would have reached a local minima introduced purely by the grammar. This issue shares strong similarities with the question of including non-feasible solutions in constraint-based evolutionary optimization, which is addressed expertly in Michalewicz (1992).

1.1 Adding State Variables to Evolved Programs

Often, when solving an iterated problem, state information that summarizes previous input states must be maintained. For instance, if the task is to recognize strings in the regular grammar I^* then the occurrence of a zero early in the string must be noted and saved until the end of the string is reached. Failure to adequately manage pertinent state information may lead to incorrect identification of strings like 0111111111111111 as belonging to the above regular language.

In genetic programming, two forms of maintaining state information can be identified. *Implicit state representation* can be realized using certain functions, such as the *progn* statement frequently employed in Koza (1992). A *progn* statement executes each statement in its set of statements in turn returning the value of the last executed statement. Crude state information can be encoded in a *progn* when using side-effecting terminals (Teller 1994). *Explicit state representation* can also be achieved by adding appropriate components to the genetic program that permit direct manipulation of memory elements. *Indexed memory* (Teller 1994) associates an array of memory locations with a genetic program and adds the functions *write* and *read* to the function set to manage the memory. The *write* function takes two arguments, one designating the memory element to modify and the other providing the new value to be stored. The *read* function takes a single argument that identifies which memory element to read. The value stored in the designated location becomes the return value of the *read* function. A number of studies have successfully employed this form of memory (Teller 1994; Andre 1995; Langdon 1995; Teller 1996; and others). In addition, Spector and Luke (1996) has investigated the advantages of “cultural transmission” of problem specific information via a similar indexed memory shared among population members.

While indexed memory does provide a viable method for introducing state into an evolved program, it does so in a rather verbose manner. Consider what must be accomplished by evolution in order to correctly store and retrieve an encoded amount of state information. Some portion of the evolved program must be invoked to set a particular memory element at the appropriate time with the appropriately encoded information while another portion of the code must be invoked at the right time and read the designated memory location in order to retrieve the stored information.

Angeline (1998) models Multiple Interacting Programs (MIPs) after symbolic expressions of dynamical systems in order to evolve behaviors that use state information. MIPs evolves not a single program within an individual but a set of symbolic equations that refer to each other’s outputs. The output of an equation is an input for all other equations. Unlike the evaluation of genetic programs using ADFs (Koza 1992), each equation in a MIPs program is evaluated only once per iteration, as in a dynamical system. Angeline (1997) argues that this method of incorporating memory into an evolved program is more efficient than the previous studies using indexed memory.

1.1 Alternatives to Tree Representations

Another popular line of investigation involves replacing the standard parse tree representation with a different form of evaluated structure. Modifying the evaluated representation is typically done either to exploit certain characteristics of the target system or to increase the evolvability of the desired class of solutions.

A good example of exploiting the properties of the target machine can be found in Nordin (1994) which evolves variable length linear structures that contain machine code. The chief advantage of this approach is the incredible speed of evaluation achieved having the individuals represented in the machine code of the target computer. In addition, a second version of this approach is compact enough to be downloaded onto a small, autonomous robot and induce appropriate controllers in real time in the actual environment (Nordin and Banzhaf 1997).

Teller and Veloso (1995) describe a system they call PADO that represents programs as a labeled graph. Structurally, the form of the representation is similar to the finite state machines evolved in evolutionary programming, however, the execution is more complex and dependent on a separate branch decision section to determine which of a set number of states to proceed to next. The PADO representation also includes indexed memory and ADFs as previously defined. This representation has been successfully applied to several problems in signal understanding.

Another genetic programming variation described in Ashlock (1997) more explicitly uses a finite state machine representation directly coupled with parse trees. GP-Automata associates a independent tree with each state of a finite state machine. When a particular state is entered, the subtree associated with that state is evaluated and the value it returns determines the next state of the system. This representation allows a sequence of program fragments to be induced that are ordered and selected by the intermediate results. Ashlock (1997) and Ashlock and Richter (1997) apply this representation to some economic decision problems.

1.1 Alternatives Operators to Subtree Crossover

As mentioned above, in genetics-based forms of evolutionary computation, crossover is regarded as the principal operation that keeps the evolutionary process in motion. The prominent role of crossover in creating solutions in these methods is supported by an operational principle called the *building block hypothesis* which states that crossover juxtaposes smaller fit building blocks of parents to create larger fit building blocks in offspring. Mutation is often regarded as a minor operator in genetic-based systems and in some genetic programs is even excluded altogether (Koza 1992; and many others). Recent work in genetic algorithms and evolutionary programming has questioned the efficacy of crossover (Fogel and Atmar 1990; Eshelman and Shafer 1993; Fogel and Stayton 1994; and others) while recent work in genetic programming (Angeline 1997; Luke and Spector 1997) has demonstrated that subtree crossover is not always the most appropriate operator.

In genetic programming, the discounting of mutation's contributions to evolving programs can be traced to an experiment in Koza (1992; pp. 607-608). In this experiment, the performance of subtree crossover is compared to a simple form of subtree mutation. In the runs using only subtree crossover 90% of the offspring were manipulated compared to only 10% of the offspring in those

using only subtree mutation. The remaining offspring in both experiments were duplicates of the selected parents. Not surprisingly, the results show a significant advantage for subtree crossover alone over subtree mutation alone with crossover solving the problem for all runs and mutation successful for only 8% of the runs. Koza (1992, pp. 608) concludes that crossover is very important for solving this problem and argues elsewhere (Koza 1995) that this experiment constitutes evidence counter to the hypothesis that subtree mutation can perform on par with subtree crossover. Obviously, the unequal frequencies in the application of mutation and crossover used in this comparison provides a significant advantage for subtree crossover. Any operator if only applied to 10% of the offspring in the course of a run would perform poorly against a comparable operator applied to 90% of the offspring. Consequently, this experiment does not adequately reflect the relative performance of subtree crossover and subtree mutation, which is more adequately demonstrated in Angeline (1997) and Luke and Spector (1997).

Chellapilla (1997) and Angeline (1998) take similar approaches to the modification of genetic programs inspired by the early work on evolving finite state machines in evolutionary programming. Both use a series of mutations geared at different structural elements of the parse tree representation. Mutations are included that alter a single positions in a tree, or swap the ordering of nodes, as well as perform more caustic functions such as reducing a randomly selected subtree to a single terminal or replacing a subtree with a new, randomly generated one. Chellapilla (1998) has even shown that this approach can out perform standard genetic programming.

Taking a significantly radical approach to the induction of programs, Schmidhuber and Salustowicz (1997) describe an evolutionary system that does not use any of the conventional operators but relies instead on a reinforcement learning technique. In PIPE, a single prototype program is evolved. Each node in the prototype program carries with it all possible symbols that can be placed at that position in a final program. Probabilities are associated with each symbol at each node specifying the chance of producing that symbol at that position. A candidate program is generated by walking through the structure one node at a time, and selecting a symbol at random biased by the probabilities for each at that node location. Once some number of candidates are created, their individual fitnesses are used to dictate how the probabilities at each position should be updated using a reinforcement learning scheme. Eventually, the probabilities converge to near certainty for all positions in the prototype program and a single program is identified.

1.1 Introns and Bloating in Evolved Programs

When evolving programs, it is often the case that the expression of the solution behaviors is significantly larger than necessary, often including large portions of code that are inconsequential to the solution. Two terms are generally used to refer to such phenomena but are often confused in the literature. Consequently, within the genetic programming community, the terms *intron* and *bloat* are often considered to refer to the same phenomena (see Langdon and Poli (1997) for example). However, these terms delineate distinct issues concerning organizations of evolved variable-length symbolic structures, both of which deserve separate consideration.

An intron, in the biological sense, indicates a portion of the DNA that is not transcribed into proteins. In essence, introns in DNA provide no direct influence on the expression of the phenotype. In evolved symbolic structures, there are two manners in which an agglomerated set of symbols

can provide no direct influence on the interpretation of the symbolic structure. A *syntactic intron* is an intron that is never transcribed by the evaluation process, usually due to some form of conditional behavior prior to that section of code. Such introns can be removed completely from the evolved structure without any alteration to the processing it encodes. Syntactic introns are transparent with respect to the parsing of the symbolic structure since the evaluator never reaches these symbols.

On the other hand, a *semantic intron* is evaluated but provides no behavioral contribution to the individual's fitness. Such introns are produced when inverse operations are present in the symbol set. For instance, a boolean structure of the form $not(not(<subtree>))$ encodes a semantic intron that twice negates the boolean value of its subtree to return the subtree's original value. These organizations may be arbitrarily complex or even distributed in different portions of the evolved structure, so long as they are perfect inverses and provide no contribution to the overall behavior under any set of inputs. Their semantic transparency, the fact that they ultimately make no contribution to the evaluated behavior of the individual, makes them unobservable in the phenotype even though they are evaluated. In a sense, a syntactic intron is transparent to the parsing of the structure while a semantic intron is transparent to the evaluation of the structure. By the definitions given here, introns are independent of the behavior encoded by the evolved symbolic structure but are relative to the semantics of the symbols used to express that behavior.

Bloat, then, can be defined as the expression of a behavior using an excessive amount of the provided symbols relative to the minimum number of those symbols required to express the same behavior. For instance, if the desired behavior is to output the value "4" then the structure "1+1+1+1" expresses that behavior with more symbols than the structure "2+2" and is thus more bloated. Bloat is defined relative to both the behavior being expressed and the language being used to express that behavior. Note that introns can be one reason for bloat in an evolved structure but are not the only form and are probably not the most problematic.

Angeline (1994) argues that introns, as defined here, may be an important part of the evolutionary process. This argument has been substantiated through experimentation in both genetic algorithms (Levenick 1991) and genetic programming (Nordin, Francone, and Banzhaf 1996). Introns can provide a buffer against the destructive effects of crossover while also supplying a source of symbols that can be recruited into new behaviors through subsequent evolution. Bloating, in general, serves no conceivable purpose to the evolutionary process and can often slow the process by forcing the evaluation of exceptionally large structures that encode simple behaviors relative to the expressiveness of the language.

Attempts to directly reduce bloat typically center on modifying the fitness function to include a term that incorporates the size of the evolving structure (e.g. Iba, de Garis, and Sato 1994). While these approaches have been shown to reduce the size of induced structures they typically do not perform as well as those methods that do not limit the dynamics of the evolutionary process.

1.1 Morphogenic Genetic Programs

Given that the distinguishing feature of all genetic programs is that they are executed, the choice of exactly what in its execution to grade for fitness is arbitrary. In some genetic programs, the operations encoded in the evolved program are interpreted as a developmental process rather than

the solution behavior. When morphogenic genetic programs (Angeline 1995) are evaluated they create another structure which is then evaluated as the solution to the task. Fitness of the original program is then derived from the fitness of the structure it creates when evaluated.

Gruau (1994) describes a morphogenic genetic program that induces a program that constructs any of a family of neural networks. The evolved programs consist of network construction commands designed so that at the completion of a program's execution a functioning neural network is produced. The created network is then tested on the intended task and its performance is used as the fitness of the creating program. Gruau (1994) describes an experiment where he evolves a program that can produce a working network for parity problems of any specified size. Given an input width, the program creates the associated parity network for that width.

Koza et al. (1997) describe a variation of Gruau's idea applied to the induction of electrical circuits. Here, programs contain specifications that construct the circuit component by component. Once constructed, the circuit is tested for the desired behavior with the constructor program's fitness is set to be proportional to how well circuit performs.

It should be noted that the additional layer of execution involved in morphogenic genetic programs make the induction more difficult. In cases like Gruau (1994) where the focus is to induce a constructor for a family of solutions this added cost is acceptable. The development step permits some adaptability in the creation process allowing the result to be more responsive to the specific situation rather than induced once and used for all situations. When only a specific end result is desired, as in Koza et al. (1997), it makes little sense to expend this additional computation if the structure can be induced directly and possibly more efficiently.

2 Conclusions

Programming is a uniquely human occupation even though the exactness of the specification required for complex behaviors can be extremely difficult for a human to achieve easily. Traditional concepts of programming have been defined with the limitations of humans in mind. Evolutionary methods are not plagued with the same limitations as humans and can be enlisted to support the construction of complex behavioral solutions ultimately amplifying the abilities of human programmers. The form of program representation most appropriate for such evolutionary search will no doubt be significantly different from that used by humans. this paper described some of the properties associated with the current trends in evolving programs.

The evolution of executable structures is currently enjoying a popularity unmatched in the history of computer science. The diversity of the current research along with the breadth of application of the techniques promises. Continued diversification and further exploration of appropriate representations and techniques is required in order to determine the best synergism between human and computer collaboration. Given the current level of enthusiasm in the field, genetic programming and the evolution of executable structures will continue at pace for the foreseeable future.

2 Bibliography

- Andre, D. (1995). "The Evolution of Agents that Build Mental Models and Create Simple Plans Using Genetic Programming", In *Proceedings of the Sixth International Conference on Genetic Algorithms*, L. Eshelman (ed.), San Francisco: Morgan Kaufmann, pp. 248-255.
- Angeline, P. J. (1994). Genetic Programming and Emergent Intelligence. In *Advances in Genetic Programming*, K. Kinnear (ed.), Cambridge, MA: MIT Press, pp. 75-96.
- Angeline, P. J. (1996). Genetic Programming's Continued Evolution. In *Advances in Genetic Programming: Volume II*, P. Angeline and K. Kinnear (eds.), Cambridge MA: MIT Press, pp. 1-20.
- Angeline, P. J. (1997a). "Comparing Subtree Crossover with Macro Mutation," In *The Sixth Conference on Evolutionary Programming*, P. Angeline, R. Reynolds, J. McDonnell and R. Eberhart (eds.), Springer-Verlag, In Press.
- Angeline, P. J. (1997b) "Subtree Crossover: Building Block Engine or Macromutation", In *Genetic Programming 1997: Proceedings of the Second Annual Conference*, J. Koza, K. Deb, M. Dorigo, D. Fogel, M. Garzon, H. Iba, and R. Riolo, Cambridge, MA: MIT Press, pp. 9-17.
- Angeline P. J. (1998) "Multiple Interacting Programs: A Representation for Evolving Complex Behaviors", *Cybernetics and Systems*, To Appear.
- Ashlock, D. (1997) "GP_Automata for Dividing the Dollar." In *Genetic Programming 1997: Proceedings of the Second Annual Conference*, J. Koza, K. Deb, M. Dorigo, D. Fogel, M. Garzon, H. Iba, and R. Riolo, Cambridge, MA: MIT Press, pp. 18-26.
- Ashlock, D. and Richter, C. (1997) "The effect of splitting populations on bidding strategies" In *Genetic Programming 1997: Proceedings of the Second Annual Conference*, J. Koza, K. Deb, M. Dorigo, D. Fogel, M. Garzon, H. Iba, and R. Riolo, Cambridge, MA: MIT Press, pp. 27-34.
- Bickle, A. S. and Bickle R. W. (1987) "Tree Structured Rules in Genetic Algorithms," in *Genetic Algorithms and their Applications, Proceedings of the Second International Conference on Genetic Algorithms*, JJ. Grefentette (ed.), Lawrence Erlbaum, Hillsdale, NJ, pp. 77-81.
- Chellapilla, K. (1997) Evolutionary Programming with Tree Mutations: Evolving Computer Programs without Crossover, In *Genetic Programming 1997: Proceedings of the Second Annual Conference*, J. Koza, K. Deb, M. Dorigo, D. Fogel, M. Garzon, H. Iba, and R. Riolo, Cambridge, MA: MIT Press, pp. 431-438.
- Chellapilla, K. (1998) "Evolving computer programs without subtree crossover," *IEEE Transactions on Evolutionary Computation*, 1 (3), pp. 209-216.
- Cramer, N. (1985) A Representation for the Adaptive Generation of Simple Sequential Programs. In *Proceedings of an International Conference on Genetic Algorithms and their Applications*, J. Grefenstette (ed.), Hillsdale, NJ: Lawrence Erlbaum.
- Dickmanns, D., Schmidhuber, J., and Winklhofer, A. (1986) "Der genetische algorithmus: Eine implementierung in Prolog," Fortgeschrittenenpraktikum, Institut für Informatik, Lehrstuhl Prof. Radig, Technische universitat Munchen, Germany.

- Eshelman, L. J. and Shaffer, J. D. (1993) Crossover's Niche, In *Proceedings of the Fifth International Conference on Genetic Algorithms*, S. Forrest (ed.), San Mateo, CA: Morgan Kaufmann, pp. 9-14.
- Fogel, D.B. (1995). *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*, Piscataway, NJ: IEEE Press.
- Fogel, D. B. and Atmar, J. W. (1990) Comparing Genetic Operators with Gaussian Mutation in Simulated Evolutionary Processes Using Linear Systems, *Biological Cybernetics*, 63, pp 111-114.
- Fogel, D. B. and Stayton, L. C. (1994). On the Effectiveness of Crossover in Simulated Evolutionary Optimization, *BioSystems*, 32, pp. 171-182.
- Fogel, D.B. (1995). *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*, Piscataway, NJ: IEEE Press.
- Fogel, L.J., Owens, A.J. and Walsh, M.J. (1966). *Artificial Intelligence through Simulated Evolution*, New York: John Wiley.
- Friedberg, R. M. (1958) "A learning machine: Part I," *IBM Journal of Research and Development*, 2, pp. 2-13.
- Friedberg R. M., Dunham, B., and North, J. H. (1959) "A learning machine: Part II," *IBM Journal of Research and Development*, 3, pp. 282-287.
- Fujiki, C. (1986) "An evaluation of Holland's genetic operators applied to a program generator," Masters Thesis, University of Idaho, Moscow, ID.
- Fujiki, C. and Dickinson, J. (1987) "Using the genetic algorithm to generate LISP source code to solve the prisoner's dilemma," in *Genetic Algorithms and their Applications, Proceedings of the Second International Conference on Genetic Algorithms*, JJ. Grefentette (ed.), Lawrence Erlbaum, Hillsdale, NJ, pp. 236-240.
- Gruau, F. (1994) "Genetic microprogramming of neural networks," In *Advances in Genetic Programming*, K. Kinnear (ed.), Cambridge, MA: MIT Press, pp. 495-518.
- Gruau, F. (1996) "On using syntactic constraints with genetic programming," In *Advances in Genetic Programming: Volume II*, P. Angeline and K. Kinnear (eds.), Cambridge MA: MIT Press, pp. 377-394.
- Haynes, T. D., Schoenefield, D. A., and Wainwright, R. L. (1996) "Type inheritance in strongly typed genetic programming," In *Advances in Genetic Programming: Volume II*, P. Angeline and K. Kinnear (eds.), Cambridge MA: MIT Press, pp. 359-376.
- Hicklin, J. F. (1986) "Application of the genetic algorithm to automatic program generation," Masters Thesis, University of Idaho, Moscow, ID.
- Holland, J.H. (1992). *Adaptation in Natural and Artificial Systems*, Ann Arbor, MI: University of Michigan Press.
- Iba, H., de Garis, H., and Sato, T. (1994) Genetic Programming Using a Minimum Description Length Principle, In *Advances in Genetic Programming*, K. Kinnear (ed.), Cambridge, MA: MIT Press, pp. 265-284.

- Koza, J. R. (1989) "Hierarchical genetic algorithms operating on populations of computer programs," Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, N. S. Sridaran (ed.), Morgan Kaufmann, San Mateo, CA, pp. 768-774.
- Koza, J. R. (1992). *Genetic Programming: On the programming of computers by means of natural selection*. Cambridge, MA: MIT Press.
- Koza, J. R. (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs*, Cambridge, MA: MIT Press.
- Koza, J. R. (1995). A Response to the ML-95 Paper Entitled "Hill Climbing Beats Genetic Search on a Boolean Circuit Synthesis Problem of Koza's," Unpublished manuscript.
- Koza, J. R., Bennett, F. H., Andre, D. Keane, M. A., and Dunlap, F. (1997) "Automated synthesis of analog electrical circuits by means of genetic programming," *IEEE Transactions on Evolutionary Computation*, 1 (2), pp. 109-128.
- Langdon, W. B. (1995) "Evolving Data Structures with Genetic Programming", In *Proceedings of the Sixth International Conference on Genetic Algorithms*, L. Eshelman (ed.), San Francisco: Morgan Kaufmann, pp. 295-302.
- Langdon, W. B. and R. Poli (1997) Fitness Causes Bloat: Mutation, In *Late Breaking Papers at the GP-97 Conference*, J. Koza (ed.), Stanford, CA: Stanford Book Store, pp. 132-140.
- Levenick, J. (1991) Inserting Introns Improves Genetic Algorithm Success Rate: Taking a Cue from Biology. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, R. Belew and L. Booker (eds.), San Mateo, CA: Morgan Kaufmann Publishers Inc., pp. 123-127.
- Luke, S. and L. Spector (1997) "A Comparison of Crossover and Mutation in Genetic Programming," In *Genetic Programming 1997: Proceedings of the Second Annual Conference*, J. Koza, K. Deb, M. Dorigo, D. Fogel, M. Garzon, H. Iba, and R. Riolo, Cambridge, MA: MIT Press, pp. 240-248.
- Michalewicz, Z. (1992) *Genetic Algorithms + Data Structures = Evolution Programs*, Springer, Berlin.
- Mitchell, M., Forrest, S. and Holland, J. (1992) The Royal Road Functions for genetic algorithms: Fitness landscapes and GA performance, In *Toward a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life*. F. Verela and P. Bourguine (eds.), Cambridge, MA: MIT Press.
- Montana, D. (1995) "Strongly typed genetic programming," *Evolutionary Computation*, 3 (2), pp. 199-230.
- Nordin, P. (1994) "A compiling genetic programming system that directly manipulates the machine code," In *Advances in Genetic Programming*, K. Kinnear (ed.), Cambridge, MA: MIT Press, pp. 311-332.
- Nordin, P. Francone, F., and Banzhaf, W. (1996) "Explicitly Defined Introns and Destructive Crossover in Genetic Programming," In *Advances in Genetic Programming: Volume 2*, P. Angeline and K. Kinnear (eds.), Cambridge, MA: MIT Press, pp. 111-134.
- Nordin, P. (1997) "An on-line method to evolve behavior and to control a miniature robot in real time with genetic program," *Adaptive Behavior*, 5 (2), pp. 107-140.

- O'Reilly and Oppacher (1995) The Troubling Aspects of a Building Block Hypothesis for Genetic Programming, In *Foundations of Genetic Algorithms 3*, L. D. Whitley and M. D. Vose (eds.), San Francisco, CA: Morgan Kaufmann, pp. 73-90.
- O'Reilly, U.-M. and F. Oppacher (1996) "A Comparative Analysis of Genetic Programming", In *Advances in Genetic Programming: Volume 2*, P. Angeline and K. Kinnear (eds.), Cambridge, MA: MIT Press, pp. 23-44.
- Rosca, J. P. (1997) "Analysis of complexity drift in genetic programming," In *Genetic Programming 1997: Proceedings of the Second Annual Conference*, J. Koza, K. Deb, M. Dorigo, D. Fogel, M. Garzon, H. Iba, and R. Riolo, Cambridge, MA: MIT Press, pp. 286-294.
- Rosca, J. P. and Ballard, D. H. (1996) "Discovery of subroutines in genetic programming," In *Advances in Genetic Programming: Volume 2*, P. Angeline and K. Kinnear (eds.), Cambridge, MA: MIT Press, pp. 177-202.
- Salustowicz, R. and Schmidhuber, J. (1997) "Probabilistic incremental program evolution," *Evolutionary Computation*, 5 (2), pp. 123-142.
- Simon, H. A. (1969) *The Sciences of the Artificial*, Cambridge, MA, MIT Press.
- Smith, S. F. (1980) "A learning system based on genetic adaptive algorithms, Ph. D. Thesis, University of Pittsburgh, Pittsburgh, PA.
- Smith, S. F. (1983) "Flexible learning of problem solving heuristics through adaptive search," *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, A. Bundy (ed.), William Kausman, Inc., Los Altos, CA, pp. 422-425.
- Spector, L. and Luke, S. (1996). "Cultural Transmission of Information in Genetic Programming", In *Genetic Programming 1996: Proceedings of the First Annual Conference*, J. Koza, D. Goldberg, D. Fogel and R. Riolo (eds.), Cambridge, MA: MIT Press, pp.209-214.
- Teller, A. (1994). "The Evolution of Mental Models", in *Advances in Genetic Programming*, K. Kinnear (ed.), Cambridge, MA: MIT Press, pp. 199-219.
- Teller, A. (1996) "Evolving Programmers: The Co-evolution of Intelligent Recombination Operators", In *Advances in Genetic Programming: Volume 2*, P. Angeline and K. Kinnear (eds.), Cambridge, MA: MIT Press, pp. 45-68.
- Teller, A. and M. Veloso (1996) "Neural Programming and an Internal Reinforcement Property", In *Late Breaking Papers at the Genetic Programming 1996 Conference*, J. Koza (ed.), Stanford, CA: Stanford University Bookstore, pp. 186-192.
- Turing, A. (1950) "Computing machinery and intelligence," In *Mind* LIX, no. 2236 (Oct. 1950), Oxford University Press, pp. 433-460.