Version 3 – Submitted November 10, 1996 to *Models of Action: Mechanisms for Adaptive Behavior* edited by Clive Wynne and John Staddon for Lawrence Erlbaum Associates, Inc, Publishers.

USING BIOLOGY TO SOLVE A PROBLEM IN AUTOMATED MACHINE LEARNING

John R. Koza
Computer Science Department
Stanford University
Margaret Jacks Hall
Stanford, California 94305 USA
E-MAIL: Koza@CS.Stanford.Edu
WWW: http://www-cs-faculty.stanford.edu/~koza/
PHONE: 415-941-0336
FAX: 415-941-9430

ABSTRACT

This chapter describes how the biological theory of gene duplication described in Susumu Ohno's provocative book, *Evolution by Means of Gene Duplication*, was brought to bear on a vexatious problem from the domain of automated machine learning.

The goal of automatic programming is to create, in an automated way, a computer program that enables a computer to solve a problem. Ideally, an automatic programming system should require that the user pre-specify little about the problem environment.

Genetic programming is a domain-independent approach to automated machine learning that attempts to evolve a computer program that solves, or approximately solves, problems. Starting with a primordial ooze of randomly generated computer programs composed of the available programmatic ingredients, genetic programming applies the principles of animal husbandry (including Darwinian selection and sexual recombination) to breed new (and often improved) populations of computer programs.

One of the undesirable aspects of many techniques of automated machine learning is that the user of the technique may be required to specify the size and shape (i.e., the architecture) of the ultimate solution to his problem before he can begin to apply the technique to his problem. Specification of the size and shape of the solution often corresponds to discovering a way to decompose the problem into useful subspaces (usually of lower dimensionality) or to discovering a congenial representation of the problem that facilitates solution of the problem. Thus, in practice, for many problems of interest, determining the size and shape of the solution may be *the problem* (or at least a substantial part of the problem).

This chapter describes how biology motivated a solution to the problem of architecture discovery for genetic programming. The resulting biologically-motivated approach enables genetic programming to automatically discover the size and shape of the solution at the same time as genetic programming is evolving a solution to the problem. This is accomplished using six new architecture-altering operations that provide a way to automatically discover, during a run of genetic programming, both the architecture and the

sequence of steps of a multi-part computer program that will solve the given problem.

1. Introduction

The goal of automatic programming is to create, in an automated way, a computer program that enables a computer to solve a problem. This goal (attributed to Arthur Samuel in the 1950s) can be stated as follows:

> How can computers learn to solve problems without being explicitly programmed?

Genetic programming is a domain-independent method for evolving a computer program for solving, or approximately solving, a problem (Koza 1989). Genetic programming is an extension of the biologically motivated *genetic algorithm* that was first described in John Holland's pioneering *Adaptation in Natural and Artificial Systems* (1975, 1992). David Goldberg (1989) and Mitchell (1996) survey current work the field of genetic algorithms.

Genetic programming starts with a primordial ooze of randomly generated computer programs composed of the available programmatic ingredients and then applies the principles of animal husbandry to breed a new (and often improved) population of programs. The breeding is done in a domain-independent way using the Darwinian principle of survival of the fittest and an analog of the naturally-occurring genetic operation of crossover (sexual recombination). The crossover operation is designed to create syntactically valid offspring computer programs from any selected pair of parental computer programs (given closure amongst the set of ingredients). Genetic programming combines the expressive

high-level symbolic representations of computer programs with the near-optimal efficiency of learning of Holland's genetic algorithm.

Genetic Programming: On the Programming of Computers by Means of Natural Selection (Koza 1992) provides evidence that genetic programming can solve, or approximately solve, a variety of problems from a variety of fields, including many benchmark problems from machine learning, artificial intelligence, control, robotics, optimization, game playing, symbolic regression, system identification, and concept learning. Recent advances in genetic programming are described in Kinnear 1994, Angeline and Kinnear 1996, and the proceedings of the annual genetic programming conferences (Koza, Goldberg, Fogel, and Riolo 1996).

The sequence of the work-performing steps of the programs being evolved by genetic programming is not specified in advance by the user. Instead, the sequence of steps is evolved by genetic programming as a result of the competitive and selective pressures of the evolutionary process and the recombinative role of crossover. However, this first book *Genetic Programming* has the limitation that the vast majority of its evolved programs are single-part (i.e., one result-producing main part, but no subroutines).

I believe that no approach to automated programming is likely to be successful on non-trivial problems unless it provides some hierarchical mechanism to exploit, *by reuse* and *parameterization*, the regularities, symmetries, homogeneities, similarities, patterns, and modularities inherent in problem environments. Subroutines do this in ordinary computer programs. Accordingly, *Genetic Programming II: Automatic Discovery of Reusable Programs* (Koza

1994a) describes how genetic programming can be extended to evolve multi-part programs consisting of a main program and one or more reusable, parameterized, hierarchically-called subprograms.

An *automatically defined function* (*ADF*) is a function (i.e., subroutine, subprogram, `DEFUN`, procedure, module) that is dynamically evolved during a run of genetic programming and which may be called by a calling program (or subprogram) that is concurrently being evolved. When automatically defined functions are being used, a program in the population consists of a hierarchy of one (or more) *reusable* function-defining branches (i.e., automatically defined functions) along with a main result-producing branch. Typically, the automatically defined functions possess one or more dummy arguments (formal parameters) and are reused with different instantiations of these arguments. As a run progresses, genetic programming evolves different subprograms in the function-defining branches, different main programs in the result-producing branch, different instantiations of the dummy arguments (formal parameters) of the automatically defined functions in the function-defining branches, and different hierarchical references between the automatically defined functions.

When automatically defined functions are being used in genetic programming, the initial random generation of the population is created so that every individual program in the population has a constrained syntactic structure consisting of a particular architectural arrangement of branches. When crossover is to be performed, a *type* is assigned to each potential crossover point in the parental computer programs either on a branch-wide basis (called *branch typing* or *like-branch typing*) or on the basis of the actual content of the subtree below each

potential crossover point (called *point typing*). Crossover is then performed in a structure-preserving way (given closure) so as to preserve the syntactic validity of all offspring (Koza 1994a).

Genetic programming with automatically defined functions has been shown to be capable of solving numerous problems. More importantly, the evidence so far indicates that, for many problems, genetic programming requires less computational effort (i.e., fewer fitness evaluations to yield a solution with a satisfactorily high probability) with automatically defined functions than without them (provided the difficulty of the problem is above a certain relatively low break-even point). Also, genetic programming usually yields solutions with smaller average overall size with automatically defined functions than without them (again provided that the problem is not too simple). That is, both learning efficiency and parsimony appear to be properties of genetic programming with automatically defined functions.

Moreover, there is also evidence that genetic programming with automatically defined functions is scalable. For a limited number of problems for which a progression of scaled-up versions was studied, the computational effort increases as a function of problem size at a *slower rate* with automatically defined functions than without them. In addition, the average size of solutions similarly increases as a function of problem size at a *slower rate* with automatically defined functions than without them. This observed scalability results from the profitable reuse of hierarchically-callable, parameterized subprograms within the overall program.

There are five major preparatory steps required before genetic programming can be applied to a particular problem, namely determining

(1) the set of terminals (i.e., the actual variables of the problem, zero-argument functions, constants) for each branch,

(2) the set of functions (e.g., primitive functions) for each branch,

(3) the fitness measure (or other arrangement for implicitly measuring fitness),

(4) the parameters to control the run, and

(5) the termination criterion and the result designation method for the run.

When automatically defined functions are added to genetic programming, it is also necessary to determine the architecture of the yet-to-be-evolved programs. The specification of the architecture consists of

(a) the number of function-defining branches in the overall program,

(b) the number of arguments (if any) possessed by each function-defining branch, and

(c) if there is more than one function-defining branch, the nature of the hierarchical references (if any) allowed between the function-defining branches.

Sometimes these architectural choices flow directly from the nature of the problem. Sometimes heuristic methods are helpful. However, in general, there is no way of knowing *a priori* the optimal (or minimum) number of automatically defined functions that will prove to be useful for a given problem, or the optimal (or sufficient) number of arguments for each automatically defined function, or the optimal (or sufficient) arrangement of hierarchical references among the automatically defined functions.

If the goal is to develop a single, unified, domain-independent approach to automatic programming that requires that the user pre-specify as little direct

information as possible about the problem, the question arises as to whether these architectural choices can be automated. Indeed, the requirement that the user predetermine the size and shape of the ultimate solution to a problem has been a bane of automated machine learning from the earliest times (Samuel 1959).

In other words, the question is whether genetic programming can be enabled to discover the architecture of a multi-part program during a run. Presumably, as the evolutionary process proceeds from generation to generation, various architectures would be dynamically created during the run of genetic programming. The new architectures would then be tested as to how well they solve the problem at hand. Certain individuals with certain architectures will prove to be more fit than others at grappling with the problem. The more fit architectures might tend to prosper, while the less fit architectures might tend to wither away. Eventually a computer program with an appropriate architecture might emerge from this process of *evolution of architecture*.

Section 2 of this chapter describes the naturally occurring processes of gene duplication and gene deletion. Section 3 describes automatically defined functions. Section 4 describes the six new architecture-altering operations. Section 5 discusses the implications of the new architecture-altering operations. Section 6 demonstrates that the problem of symbolic regression of the Boolean even-5-parity function can be solved while the architecture is being simultaneously evolved by showing an example of an actual run. Section 7 compares the computational effort required for five different ways of solving the parity problem using genetic programming, including the new way described in this chapter involving the evolution of architecture. Section 8 is the conclusion.

2. Gene Duplication and Deletion in Nature

Nature points the way to perform the *evolution of architecture* that would enable genetic programming to dynamically discover the program architecture for solving a problem. A change in the architecture of a multi-part computer program during a run of genetic programming corresponds to a change in genome structure in the natural world. Therefore, it seems appropriate to consider the different ways that a genomic structure may change in nature.

In nature, sexual recombination (crossover) exchanges alleles (gene values) at particular locations (loci) along the chromosome (a molecule of DNA). The DNA then controls the manufacture of various proteins that determine the structure, function, and behavior of the living organism (Stryer 1988). The resulting organism then spends its life attempting to grapple with its environment. Some organisms in a given population do better than others in that they survive to the age of reproduction, produce offspring, and thereby pass on all or part of their genetic make-up to the next generation of the population. Over a period of time and many generations, the population as a whole evolves so as to give increasing representation to traits (and, more importantly, co-adapted combinations of traits) that contribute to survival of the organism to the age of reproduction and that facilitate large numbers of offspring. This process, which Charles Darwin (1859) called *natural selection*, tends to evolve near-optimal (perhaps even optimal) co-adapted sets of alleles in the chromosomes of the organism (given its environment).

John Holland's pioneering *Adaptation in Natural and Artificial Systems* (1975) described how an analog of the naturally-occurring evolutionary process can be applied to solving artificial problems using what is now called the *genetic*

algorithm. Before applying the genetic algorithm to the problem, the user designs an artificial chromosome of a certain size and shape and then defines a mapping (encoding) between the points in the search space of the problem and the artificial chromosome. For example, in applying the genetic algorithm to a multidimensional optimization problem (where the goal is to find the global optimum of an unknown multidimensional function), the artificial chromosome is often a linear character string (modeled directly after the linear string of information found in DNA). A specific location (a gene) along this artificial chromosome is associated with each of the variables of the problem. Character(s) appearing at a particular location along the chromosome denote the value of a particular variable (i.e., the gene value or allele). Each individual in the population has a fitness value (which, for a multidimensional optimization problem, is the value of an unknown function). The genetic algorithm then manipulates a population of such artificial chromosomes (usually starting from a randomly-created initial population of strings) using the operations of reproduction, crossover, and mutation. Individuals are probabilistically selected to participate in these genetic operations based on their fitness. That is, individuals with better fitness are more likely to be selected in the genetic operations. The goal of the genetic algorithm in a multidimensional optimization problem is to find an artificial chromosome which, when decoded and mapped back into the search space of the problem, corresponds to a globally optimum (or near-optimum) point in the search space of the problem. The probabilistic aspect of the genetic algorithm is important. The best individuals are not guaranteed to be selected. Even poor individuals in the population are sometimes selected.

Both the natural and artificial evolutionary processes described above indicate how a globally optimum combination of alleles (gene values) within a fixed-size chromosome can be discovered by means of evolution. However, in both the natural and artificial processes described above, the crossover operation merely exchanges alleles (gene values) at particular locations along an already-existing chromosomal structure of fixed size and shape. The above description does not address the question of how genome lengths change during the course of evolution. Neither does the above description address the question of how totally new structures, new functions, new behaviors, and new species arise.

In nature, there is not only short-term optimization of alleles in their fixed locations within a fixed-size chromosome, but long-term emergence of new proteins (which, in turn, create new structures, functions, and behaviors and thereby sometimes create new and more complex organisms). The emergence of new proteins alters the architecture of the chromosome. Indeed, genome lengths in nature have generally increased with the emergence of new and more complex organisms (Dyson and Sherratt 1985, Brooks Low 1988). In genetic algorithms, a change in the architecture and length of a chromosome corresponds to a dynamic alteration, during a run of the algorithm, of the user-created mapping (the encoding and decoding) between points from the search space of the problem and instances of the artificial chromosome. In genetic programming, a change in the architecture of the evolving program corresponds to a change in the number of automatically defined functions, the number of arguments possessed by each automatically defined function in an overall program, and in the pattern of hierarchical references among the automatically defined functions.

As I considered how to solve the problem of how to evolve the architecture of the multi-part computer program during a run of genetic programming, I realized that an analogous mechanism must operating in nature. Therefore, it seemed appropriate to examine the mechanism by which genome structure is altered in nature over the course of millions of years of evolution.

Gene duplications are rare and unpredictable events in the evolution of genomic sequences. In gene duplication, there is a duplication of a lengthy portion of the linear string of nucleiotide bases of the DNA in the living cell. After a sequence of bases that code for a particular protein is duplicated in the DNA, there are two identical ways of manufacturing the same protein. Thus, there is no immediate change in the proteins that are manufactured as a result of a gene duplication even though the genomic structure has changed.

Over time, however, some other genetic operation, such as mutation or crossover, may change one or the other of the two identical genes. Over short periods of time, the changes accumulating in the changed gene may be of no practical effect or value. As long as one of the two genes remains unchanged, the original protein manufactured from the unchanged gene continues to be manufactured and the structure and behavior of the organism involved may continue as before. The changed gene is simply carried along in the DNA from generation to generation.

Natural selection exerts a powerful force in favor of maintaining a gene that encodes for the manufacture of a protein that is important for the survival and successful performance of the organism. However, after a gene duplication has occurred, there is no disadvantage associated with the loss of the *second* way of

manufacturing the original protein. Consequently, natural selection usually exerts little or no pressure to maintain a second way of manufacturing a particular protein. Over a period of time, the second gene may accumulate additional changes and diverge more and more from the original gene. Eventually the changed gene may lead to the manufacture of a distinctly new and different protein that actually does affect the structure and behavior of the living thing in some advantageous or disadvantageous way. When a changed gene leads to the manufacture of a viable and advantageous new protein, natural selection again works to preserve that new gene.

Ohno's *Evolution by Gene Duplication* (1970) corrects the mistaken notion that natural selection is a mechanism for promoting change. Instead, Ohno emphasizes the essentially conservative role of natural selection in the evolutionary process:

> "...the true character of natural selection ... is not so much an advocator or mediator of heritable changes, but rather it is an extremely efficient policeman which conserves the vital base sequence of each gene contained in the genome. As long as one vital function is assigned to a single gene locus within the genome, natural selection effectively forbids the perpetuation of mutation affecting the *active* sites of a molecule." (Emphasis in original).

Ohno further points out that ordinary point mutation and crossover are insufficient to explain major evolutionary changes.

> "...while allelic changes at already existing gene loci suffice for racial differentiation within species as well as for adaptive radiation from an

> immediate ancestor, they cannot account for large changes in evolution, because large changes are made possible by the acquisition of new gene loci with previously non-existent functions."

Ohno continues,

> "Only by the accumulation of *forbidden* mutations at the *active* sites can the gene locus change its basic character and become a new gene locus. An escape from the ruthless pressure of natural selection is provided by the mechanism of gene duplication. By duplication, a redundant copy of a locus is created. Natural selection often ignores such a redundant copy, and, while being ignored, it accumulates formerly forbidden mutations and is reborn as a new gene locus with a hitherto non-existent function." (Emphasis in original).

Ohno concludes,

> "Thus, gene duplication emerges as the major force of evolution."

Ohno's provocative thesis is supported by the discovery of pairs of proteins with similar sequences of DNA and similar sequences of amino acids, but different functions. Examples include trypsin and chymotrypsin; the protein of microtubules and actin of the skeletal muscle; myoglobin and the monomeric hemoglobin of hagfish and lamprey; myoglobin used for storing oxygen in muscle cells and the subunits of hemoglobin in red blood cells of vertebrates; and the light and heavy immunoglobin chains (Nei 1987, Maeda and Smithies 1986, Dyson and Sherratt 1985, Brooks Low 1988, Patthy 1991, Go 1991, Hood and Hunkapiller 1991). For the *Escherichia coli* bacteria, a relatively simple organism, it is known that more than 30% of its proteins are the result of gene

duplications (Lazcano and Miller 1994; Riley 1993). These proteins include its DNA polymerases, dehydrogenases, ferredoxins, glutamine synthetases, carbamoyl-phosphate synthetases, F-type ATPases, and DNA topoisomerases.

The midge, *Chironomus tentans*, provides an additional example of gene duplication (Galli and Wislander 1993, 1994). In particular, we focus our attention on the particular contiguous sequence containing 3,959 nucleiotide bases of the DNA of this midge that is archived under accession number X70063 in the European Molecular Biology Laboratory (EMBL) database and the Gen Bank database. The 732 nucleiotide bases located at positions 918–1,649 of the 3,959 bases of the DNA sequence involved become expressed as a protein containing 244 (i.e., one third of 732) amino acid residues. The 759 nucleiotide bases at positions 2,513–3,271 become expressed as a protein containing 253 residues. The 732-base subsequence is called the "*C. tentans* Sp38–40.A" gene and the 759-base subsequence is called "*C. tentans* Sp38–40.B." The bases of DNA before position 918, the bases between positions 1,650 and 2,612, and the bases after position 3,371 of this sequence of length 3,959 do not become expressed as any protein.

Both the "A" and the "B" proteins are secreted from the midge's salivary gland to form two similar, but different, kinds of water-insoluble fibers. The two kinds of fibers are, in turn, spun into one of two similar, but different, kinds of tubes. One tube is for larval protection and feeding while the other tube is for pupation (the stage in the development of an insect in which it lies in repose and from which it eventually emerges in the winged form).

Table 1 shows the bases of DNA in positions 900 through 3,399 of the 3,959 nucleiotide bases of X70063. In the DNA sequence, A represents the nucleiotide base adenine, C represents cytosine, G represents guanine, and T represents thymine. Each group of three consecutive bases (a codon) of DNA becomes expressed as one of the 20 amino acid residues of the protein. The letters A, T, and G appearing at positions 918, 919, and 920, respectively in this reading frame, of the DNA sequence are translated into the amino acid residue methionine (denoted by the single letter M using the 20-letter coding for amino acid residues in proteins). Thus, methionine is the first amino acid residue (i.e., N-terminal) of protein "A." Positions 921, 922, and 923 of the DNA contain the bases A, G, and A, respectively, and these three bases, in this reading frame, are translated into arginine (an amino acid residue denoted by the letter R). Thus, arginine is the second amino acid residue of protein "A" and the protein sequence begins with the residues M and R. The DNA up to position 1,649 encodes the first protein. Positions 1,647, 1648, and 1,649 code for the amino acid resident lysine (denoted by the letter K). Thus, lysine is the last (244th) residue (i.e., C-terminal) of protein "A."

Table 2 shows the 244 amino acid residues of the *C. tentans* Sp38–40.A protein.

Table 3 shows the 253 amino acid residues of the *C. tentans* Sp38–40.b protein.

The two proteins are similar, but different. For example, the first 14 amino acid residues are identical. Residue 15 of the "A" protein is phenylalanine (F), while the residue 15 of the "B" protein is leucine (L), a chemically similar amino

acid. Residues 16–50 are identical. Residue 51 of the "A" protein is glutamic acid (E), while residue 51 of the "B" protein is Aspartic acid (D). Both D and E are similar in that both are electrically negatively charged residues at normal pH values. However, for some positions, such as 76, the amino acid residues (T and A) are not chemically or electrically similar.

If we now read from the end of each protein, we see that the last few residues of each protein are identical. Since the proteins are of different length, identification of the similarity between the two protein sequences requires aligning the two proteins in some way. Protein alignment algorithms, such as the Smith-Waterman algorithm (Smith and Waterman 1981), provide a way to align two proteins and to measure the degree of similarity or dissimilarity between two proteins. The Smith-Waterman algorithm is a progressive alignment method employing dynamic programming based on a scoring algorithm. Since the proteins being aligned are typically of different lengths, gaps may be introduced (and then lengthened) in an attempt to best align the residues making up the proteins. A penalty is assessed to open a gap (5 here) and another penalty is assessed to lengthen a gap (25 here). An additional penalty is assessed when one residue disagrees with another. This penalty is smaller for substitutions involving evolutionarily-close amino acid residues. The PAM-250 ("Percentage of Accepted point Mutations") matrix is used to reflect the likelihood of one amino acid residue being mutated into another. The overall scoring algorithm performs a tradeoff employing dynamic programming between the penalties assessed by the PAM-250 matrix, the gap-opening penalty, and the gap-lengthening penalty.

The Smith-Waterman algorithm has been implemented in GeneWorks, a software package available from Intelligenetics Inc. of Mountain View, California.

Table 4 shows the alignment of the *C. tentans* Sp38–40.A protein and the *C. tentans* Sp38–40.B protein. Identical residues are boxed. The alignment shows that there is 8l% identity between the two protein sequences. As can be seen, the first disagreement between the two aligned sequences occurs at position 15 and the second occurs at residue 51. The first gap is introduced at residue 112 where the "A" protein has an alanine (A) residue. A gap of length 3 is introduced at positions 147, 148, and 149 where the "A" protein has three proline (P) residues. Note that this alignment recognizes the identity between the last five residues of the two proteins. This alignment has a total cost of 265.

Galli and Wislander (1993) point out that these two similar proteins arise as a consequence of a gene duplication. Immediately after the gene duplication occurred at some time in the distant past, there were two identical copies of the duplicated sequence of DNA. Over a period of millions of years since the initial gene duplication, additional mutations accumulated so that the two proteins are now only 81% identical (after alignment). More importantly, the two proteins now perform different (but similar) functions in the midge.

More complex organisms have a general tendency to have more expressed proteins, more different kinds of structures, more complex structures, perform more different functions, and have longer genomes (Dyson and Sherratt 1985). The rise of new functions as a consequence of gene duplication is consistent with the observed longer genomes of more complex organisms.

Gene deletion also occurs in nature. In gene deletion, there is a deletion of a portion of the linear string of nucleiotide bases that would otherwise be translated and manufactured into work-performing proteins in the living cell. After a gene deletion occurs, some particular protein that was formerly manufactured will no longer be manufactured and there may be some change in the structure or behavior of the biological entity. The absence of the protein may then affect the structure and behavior of the living thing in some advantageous or disadvantageous way. If the deletion is advantageous, natural selection will tend to perpetuate the change, but if the deletion is disadvantageous, natural selection will tend to lead to the extinction of the change.

3. Automatically Defined Function in Genetic Programming

Automatically defined functions (ADFs) are the analog of subroutines in the genetic programming process.

When automatically defined functions are being used, each program in the population contains one or more function-defining branches (each defining an automatically defined function) and one main result-producing branch. The automatically defined functions can perform arithmetic, conditional, and other types of operations, define constants, define subsets, and so forth. In addition, for certain problems, there may be other problem-specific types of branches (such as iteration-performing branches and iteration-terminating branches).

Figure 1 shows an overall program consisting of one two-argument automatically defined function (called `ADF0` here) and one result-producing branch (`RPB`). The argument map describes the architecture of a multi-part program in terms of the number of its function-defining branches and the number

of arguments that they each possess. The *argument map* of the set of automatically defined functions belonging to an overall program is the list containing the number of arguments possessed by each automatically defined function in the program. The argument map for the overall program in figure 2 is {2} because there is one function-defining branch that takes two arguments.

The program in figure 2 contains architecture-defining points of the following types:

(1) the PROGN (labeled 400) appearing as the top-most point of the overall program,

(2) a DEFUN (labeled 410) as the top-most point of the function-defining branch,

(3) a name (i.e. ADF0 labeled 411) appearing as the first argument below the DEFUN,

(4) the function LIST (labeled 412) appearing as the second argument of the DEFUN,

(5) dummy arguments (such as ARG0 and ARG1 labeled as 413 and 414, respectively) appearing below LIST,

(6) the VALUES (labeled 419) of the function-defining branch appearing as the third argument of the DEFUN, and

(7) the VALUES (labeled 470) of the result-producing branch appearing as the final argument of PROGN.

If the program in figure 2 were to have more than one automatically defined function, there would be additional occurrences of items (2), (3), (4), (5), and (6) for each additional function-defining branch.

These architecture-defining points were called "invariant points" in *Genetic Programming II* because they were not subject to alteration by crossover or mutation. However, this terminology becomes obsolete with the introduction of the architecture-altering operations described herein.

The program in figure 2 also contains work-performing points. These work-performing points are the bodies of the result-producing branch and the function-defining branch(es).

The work-performing points of figure 2 include

(1) the five points labeled 420, 421, 422, 423, and 424 that are found below the VALUES (labeled 419) in the function-defining branch, and

(2) the 11 points starting with the AND (labeled 480) that are found below the VALUES (labeled 470).

These work-performing points were called "noninvariant points" in *Genetic Programming II* because these points represented the sequence of steps of the to-be-evolved computer program and because they were almost always different from branch to branch within a program and from program to program within the population. Again, this terminology becomes obsolete with the introduction of the architecture-altering operations described herein.

The result-producing branch may invoke all, some, or none of the automatically defined functions that are present within the overall program. The result-producing branch does not contain dummy arguments (formal parameters). The result-producing branch typically contains the actual variables of the problem (e.g., D0, D1, D2, etc. here).

The value returned by the overall program consists of the value returned by the result-producing branch.

The automatically defined functions of a particular overall program are usually named sequentially as `ADF0`, `ADF1`, etc.

The automatically defined functions typically each possess a certain number of dummy arguments (formal parameters). Here, `ADF0` possesses two dummy arguments, `ARG0` and `ARG1`. Typically, the actual variables do not appear in the function-defining branches.

If the overall program has more than one automatically defined function, there may (or may not) be hierarchical references between function-defining branches. For example, the function-defining branch of an overall program may be allowed to refer (non-recursively) to all other previously-defined (i.e., lower numbered) function-defining branches.

References within a particular program to an automatically defined function are to the automatically defined function belonging to that particular program.

Actions (with side effects) may be performed within the function-defining branches, the result-producing branches, or both.

When automatically defined functions are being used, the initial random generation of the population must be created so that each individual overall program in the population has the intended constrained syntactic structure. The constrained syntactic structure in figure 1 calls for one result-producing branch and one function-defining branch. The function-defining branch for `ADF0` is a random composition of functions from the function set, $\mathcal{F}_{adf}$, and terminals from the terminal set, $\mathcal{T}_{adf}$. Here the function set, $\mathcal{F}_{adf}$, consists of the two-argument

Boolean functions AND, OR, NAND, and NOR. The terminal set, $\mathcal{T}_{rpb}$, of the function-defining branch consists of the two dummy arguments (formal parameters), ARG0 and ARG1. The result-producing branch is a random composition of functions from the function set, $\mathcal{F}_{rpb}$, and terminals from the terminal set, $\mathcal{T}_{rpb}$. In figure 2, the function set, $\mathcal{F}_{rpb}$, of the result-producing branch consists of the two-argument Boolean functions AND, OR, NAND, and NOR as well as the now-defined automatically defined function, ADF0. The terminal set, $\mathcal{T}_{rpb}$, of the result-producing branch consists of the five actual variables of the problem (i.e., D0, D1, D2, etc.).

Execution of genetic programming consists of the following steps. The six operations appearing as items (2)(c)(iii) through (2)(c)(ix) are the new architecture-altering operations described in detail in a later section below.

The steps for executing genetic programming are as follows:

(1) Generate an initial random population (generation 0) of computer programs.

(2) Iteratively perform the following sub-steps until the termination criterion of the run has been satisfied:

(a) Execute each program in the population and assign it (explicitly or implicitly) a fitness value according to how well it solves the problem.

(b) Select program(s) from the population to participate in the genetic operations in (c) below.

(c) Create new program(s) for the population by applying the following genetic operations.

(i)*Reproduction*: Copy an existing program to the new population.

(ii)*Crossover*: Create new offspring program(s) for the new population by recombining randomly chosen parts of two existing programs.

(iii) *Mutation*. Create one new offspring program for the new population by randomly mutating a randomly chosen part of one existing program.

(iv) *Branch duplication*:

(v) *Argument duplication*:

(vi) *Branch deletion*:

(vii) *Argument deletion*:

(viii) *Branch Creation*:

(ix) *Argument creation*:

(3) After satisfaction of the termination criterion (which usually includes a maximum number of generations to be run as well as a problem-specific success predicate), the single best computer program in the population produced during the run (the best-so-far individual) is designated as the result of the run. This result may (or may not) be a solution (or approximate solution) to the problem.

4. Six New Architecture-Altering Genetic Operations

The six new architecture-altering genetic operations provide a new way of determining the architecture of a multi-part program. When these operations are performed during a run of genetic programming, the architecture of the participating individuals changes during the run. Meanwhile, the Darwinian selection and the reproduction operation continues to favor the more fit

individuals in the population to be modified by the usual operations of crossover and mutation.

4.1. Branch Duplication

The operation of *branch duplication* duplicates one of the branches of a program in the following way:

(1) Select a program from the population to participate in this operation.

(2) Pick one of the function-defining branches of the selected program as the branch-to-be-duplicated.

(3) Add a uniquely-named new function-defining branch to the selected program, thus increasing, by one, the number of function-defining branches in the selected program. The new function-defining branch has the same argument list and the same body as the branch-to-be-duplicated.

(4) For each occurrence of an invocation of the branch-to-be-duplicated anywhere in the selected program (e.g., the result-producing branch or any other branch that invokes the branch-to-be-duplicated), randomly choose either to leave that invocation unchanged or to replace that invocation with an invocation of the newly created function-defining branch.

The step of selecting a program for this operation (and all the other new operations described herein) is performed probabilistically on the basis of fitness, so that a program that is more fit has a greater probability of being selected to participate in the operation than a less fit program.

Figure 2 shows the program resulting after applying the operation of branch duplication to the program in figure 1 (consisting of one two-argument automatically defined function and one result-producing main branch). Specifically, the function-defining branch 410 of figure 1 defining `ADF0` (also

shown as 510 of figure 2) is duplicated and a new function-defining branch (defining ADF1) appears in figure 2.

There are two occurrences of invocations of the branch-to-be-duplicated, ADF0, in the result-producing branch of the selected program, namely ADF0 at 481 and 487 of figure 1. For each of these two occurrences, a random choice is made to either leave the occurrence of ADF0 unchanged or to replace it with the newly created ADF1. For the first invocation of ADF0 at 481 of figure 1, the choice is randomly made to replace ADF0 481 of figure 1 with the ADF1 581 in figure 2. The arguments for the invocation of ADF1 581 in figure 2 are D1 582 and D2 583 (i.e., they are identical to the arguments D1 482 and D2 483 for the invocation of ADF0 at 481 of the original program in figure 1). For the second invocation of ADF0 at 487 of figure 1, the choice is randomly made to leave ADF0 unchanged in figure 2.

Because the duplicated new function-defining branch is identical to the previously existing function-defining branch (except for the name ADF1 at 541 in figure 2) and because ADF1 is invoked with the same arguments as ADF0 had been invoked, this operation is a semantics-preserving operation in that the operation does not affect the value returned by the overall program.

The operation of branch duplication can be interpreted as a *case splitting*. After the branch duplication, the result-producing branch invokes ADF0 at 587 and ADF1 at 581 of figure 2. ADF0 and ADF1 can be viewed as separate procedures for handling the two separate newly-created subproblems (cases).

Subsequent genetic operations may alter one or both of these two presently-identical function-defining branches and these subsequent changes to lead to a

divergence in structure and behavior. This divergence may be interpreted as a *specialization* or *refinement*. That is, once `ADF0` and `ADF1` diverge, `ADF0` can be viewed as a specialization for handling for subproblem (case) associated with its invocation by the result-producing branch. Similarly, `ADF1` can be viewed as a specialization for handling its subproblem (case).

The operation of branch duplication as defined above (and all the other new operations described herein) always produce a syntactically valid program.

Analogs of the naturally occurring operation of gene duplication have been previously used with genetic algorithms operating on character strings and with other evolutionary algorithms. Holland (1975, page 116) suggested that intrachromosomal gene duplication might provide a means of adaptively modifying the effective mutation rate by making two or more copies of a substring of adjacent alleles. Cavicchio (1970) used intrachromosomal gene duplication in early work on pattern recognition using the genetic algorithm. Gene duplication is implicitly used in the messy genetic algorithm (Goldberg, Korb, and Deb 1989). Lindgren (1991) analyzed the prisoner's dilemma game using an evolutionary algorithm that employed an operation analogous to gene duplication applied to chromosome strings. Gruau (1994) used genetic programming to develop a clever and innovative technique to evolve the architecture of a neural network at the same time as the weights are being evolved.

4.2. Argument Duplication

The operation of *argument duplication* duplicates one of the dummy arguments (format parameters) in one of the automatically defined functions of a program in the following way:

(1) Select a program from the population to participate in this operation.

(2) Pick one of the function-defining branches of the selected program.

(3) Choose one of the arguments of the picked function-defining branch of the selected program as the argument-to-be-duplicated.

(4) Add a uniquely-named new argument to the argument list of the picked function-defining branch of the selected program, thus increasing, by one, the number of arguments in its argument list.

(5) For each occurrence of the argument-to-be-duplicated anywhere in the body of picked function-defining branch of the selected program, randomly choose either to leave that occurrence unchanged or to replace that occurrence with the new argument.

(6) For each occurrence of an invocation of the picked function-defining branch anywhere in the selected program, identify the argument subtree corresponding to the argument-to-be-duplicated and duplicate that argument subtree in that invocation, thereby increasing, by one, the number of arguments in the invocation.

Because the function-defining branch containing the duplicated argument is invoked with an identical copy of the previously existing argument, the effect of this operation is to leave unchanged the value returned by the overall program.

Just as the operation of branch duplication was interpreted as a case splitting, the operation of argument duplication can be similarly interpreted. After the argument duplication, the result-producing branch invokes `ADF0` with a new third argument. The particular instantiations of the second and third arguments in each invocation of `ADF0` provide potentially different ways of handling the two

separate subproblems (cases). Once the second and third arguments diverge, this divergence may be interpreted as a specialization or refinement.

4.3. Branch Deletion

The operation of branch deletion deletes one of the automatically defined functions of a program in the following way:

(1) Select a program from the population to participate in this operation.

(2) Pick one of the function-defining branches as the branch-to-be-deleted.

(3) Delete the branch-to-be-deleted from the selected program, thus decreasing, by one, the number of branches in the selected program.

(4) For each occurrence of an invocation of the branch-to-be-deleted anywhere in the selected program, replace the invocation of the branch-to-be-deleted with an invocation of a surviving branch (described below).

When a function-defining branch is deleted, the question arises as to how to modify invocations of the branch-to-be-deleted by the other branches of the overall program. One alternative (called *branch deletion by consolidation*) involves identifying a suitable second function-defining branch of the overall program as the surviving branch and replacing (consolidating) the branch-to-be-deleted with the surviving branch in each invocation of the branch-to-be-deleted. Branch deletion by consolidation can be interpreted as a way to achieve generalization in a problem-solving procedure. A second alternative (called *branch deletion with random regeneration*) is to randomly generate new subtrees composed of the available functions and terminals in lieu of an invocation of the branch-to-be-deleted. A third alternative (called *branch deletion by macro expansion*) is a semantics-preserving approach that involves inserting the entire

body of the branch-to-be-deleted for each instance of an invocation of that branch.

Both the argument duplication and the branch duplication operations create larger programs. The operations of argument deletion and branch deletion (described below) can create smaller programs and can balance the growth that would otherwise occur (provided the alternative of argument deletion by macro expansion is not used).

4.4. Argument Deletion

The operation of *argument deletion* deletes one of the arguments to one of the automatically defined functions of a program in the following way:

(1) Select a program from the population to participate in this operation.

(2) Pick one of the function-defining branches of the selected program.

(3) Choose one of the arguments of the picked function-defining branch of the selected program as the argument-to-be-deleted.

(4) Delete the argument-to-be-deleted from the argument list of the picked function-defining branch of the selected program, thus decreasing, by one, the number of arguments in the argument list.

(5) For each occurrence of an invocation of the picked function-defining branch anywhere in the selected program, delete the argument subtree in that invocation corresponding to the argument-to-be-deleted, thereby decreasing, by one, the number of arguments in the invocation.

(6) For each occurrence of the argument-to-be-deleted in the body of the picked function-defining branch, replace the argument-to-be-deleted with a surviving argument.

The operation of argument deletion may be viewed as a *generalization* in that some information that was once considered in executing a procedure is now ignored.

When an argument is deleted, references to the argument-to-be-deleted are modified by using *argument deletion by consolidation*, *argument deletion with random regeneration*, or *argument deletion by macro expansion*.

4.5. Branch Creation

The operation of branch creation creates a new automatically defined function (ADF) within an overall program in the following way:

(1) Select a program from the population to participate in this operation.

(2) Pick a point in the body of one of the function-defining branches or result-producing branches of the selected program. This picked point will become the top-most point of the body of the branch-to-be-created.

(3) Starting at the picked point, begin traversing the subtree below the picked point in a depth-first manner.

(4) As each point below the picked point in the selected program is encountered during the traversal, make a determination as to whether to designate that point as being the top-most point of an argument subtree for the branch-to-be-created. If such a designation is made, no traversal is made of the subtree below that designated point. The depth-first traversal continues and this step (4) is repeatedly applied to each point encountered during the traversal so that when the traversal of the subtree below the picked point is completed, zero points, one point, or more than one point are so designated during the traversal.

(5) Add a uniquely-named new function-defining branch to the selected program. The argument list of the new branch consists of as many consecutively-numbered dummy variables (formal parameters) as the number of points that were designated during the depth-first traversal. The body of the new branch consists of a modified copy of the subtree starting at the picked point. The modifications to the copy are made in the following way: For each point in the copy corresponding to a point designated during the traversal of the original subtree, replace the designated point in the copy (and the subtree in the copy below that designated point in the copy) by a unique dummy variable. The result is a body for the new function-defining branch that contains as many uniquely named dummy variables as there are dummy variables in the argument list of the new function-defining branch.

(6) Replace the picked point in the selected program by the name of the new function-defining branch. If no points below the picked point were designated during the traversal, the operation of branch creation is now completed.

(7) If one or more points below the picked point were designated during the traversal, the subtree below the just-inserted name of the new function-defining branch will be given as many argument subtrees as there are dummy arguments in the new function-defining branch in the following way: For each point in the subtree below the picked point designated during the traversal, attach the designated point and the subtree below it as an argument to the function whose name was just inserted in the new function-defining branch.

Several different methods may be used to determine how to designate a point below the picked point during the depth-first traversal described above.

The operation of branch creation is similar to, but different than, the compression (module acquisition) operation described by Angeline and Pollack (1994).

4.6. Argument Creation

The operation of argument creation creates a new dummy argument (formal parameter) within a function-defining branch of an overall program in the following way:

(1) Select a program from the population to participate in this operation.

(2) Pick a point in the body of a function-defining branch of the selected program.

(3) Add a uniquely-named new argument to the argument list of the picked function-defining branch for the purpose of defining the argument-to-be-created.

(4) Replace the picked point (and the entire subtree below it) in the picked function-defining branch by the name of the new argument.

(5) For each occurrence of an invocation of the picked function-defining branch anywhere in the selected program, add an additional argument subtree to that invocation. In each instance, the added argument subtree consists of a modified copy of the picked point (and the entire subtree below it) from the picked function-defining branch. The modification is made in the following way: For each dummy argument in a particular added argument subtree, replace the dummy argument with the entire argument subtree of that invocation corresponding to that dummy argument.

5. Implications of the Architecture-Altering Operations

The six new architecture-altering operations can be viewed from five perspectives.

First, the new architecture-altering operations provide a new way to solve the problem of determining the architecture of the overall program in the context of genetic programming with automatically defined functions.

Second, the new architecture-altering operations provide an automatic implementation of the ability to specialize and generalize in the context of automated problem-solving.

Third, the new architecture-altering operations, in conjunction with automatically defined functions, provide a way to automatically and dynamically change the representation of the problem while simultaneously solving the problem.

Fourth, the new architecture-altering operations, in conjunction with automatically defined functions, provide a way to automatically and dynamically decompose problems into subproblems and then automatically solve the overall problem by assembling the solutions of the subproblems into a solution of the overall problem.

Fifth, the new architecture-altering operations, in conjunction with automatically defined functions, provide a way to automatically and dynamically discover useful subspaces (usually of lower dimensionality than that of the overall problem) and then automatically assemble a solution of the overall problem from solutions applicable to the individual subspaces.

In addition, the new architecture-altering operations affect the implementation of genetic programming with regard to the creation of the initial random population and the crossover operation, as described below.

5.1. Creation of the Initial Population

When automatically defined functions are being used, the initial random generation of the population must be created so that each individual overall program in the population has the intended constrained syntactic structure. For example, figure 1 shows a program for which the constrained syntactic structure calls for one result-producing branch and one function-defining branch. The function-defining branch for ADF0 is a random composition of functions from the function set, $\mathcal{F}_{adf}$, and terminals from the terminal set, $\mathcal{T}_{adf}$. Here the function set, $\mathcal{F}_{adf}$, consists of the two-argument Boolean primitive functions AND, OR, NAND, and NOR. The terminal set, $\mathcal{T}_{rpb}$, of the function-defining branch consists of the two dummy arguments (formal parameters), ARG0 and ARG1. The result-producing branch is a random composition of functions from the function set, $\mathcal{F}_{rpb}$, and terminals from the terminal set, $\mathcal{T}_{rpb}$. In the result-producing branch of figure 1, the function set, $\mathcal{F}_{rpb}$, consists of the two-argument Boolean primitive functions AND, OR, NAND, and NOR as well as the now-defined automatically defined function, ADF0. The terminal set, $\mathcal{T}_{rpb}$, of the result-producing branch consists of the five actual variables (i.e., D0, D1, D2, D3, D4).

When the architecture-altering operations are used, the initial population of programs may be created in any one of three ways. One possibility (called the "minimalist approach") is that each multi-part program in the population at generation 0 has a uniform architecture with exactly one automatically defined function possessing a minimal number of arguments appropriate to the problem. A second possibility (called "the big bang") is that each program in the population has a uniform architecture with no automatically defined functions (i.e., only a result-producing branch). This approach relies on the operation of branch creation to create multi-part programs in such runs. A third possibility is that the population at generation 0 is architecturally diverse (as described in Koza 1994a).

5.2. Structure-Preserving Crossover

In the crossover operation in genetic programming, a crossover point is randomly and independently chosen in each of two parents and genetic material from one parent is then inserted into a part of the other parent to create an offspring. A population may be architecturally diverse either because it was initially created with architectural diversity (as described above) or because the six new architecture-altering genetic operations (described below) create a diversity of new architectures during the run. Structure-preserving crossover with point typing (as described in Koza 1994a) permits robust recombination while simultaneously guaranteeing that any pair of architecturally different parents will produce syntactically and semantically valid offspring.

If the population is architecturally diverse, the parents selected to participate in the crossover operation will often possess different numbers of automatically defined functions. Moreover, an automatically defined function with a certain name (e.g., ADF2) belonging to one parent will often possess a different number of arguments than the same-named automatically defined function belonging to the other parent (if indeed ADF2 is present at all). After a crossover is performed, each call to an automatically defined function actually appearing in the crossover fragment from the contributing parent will no longer refer to the automatically defined function of the contributing parent, but instead will refer to the same-named automatically defined function of the receiving parent.

Thus, we must redefine the crossover operation when it is employed in an architecturally diverse population.

When automatically defined functions are involved, each program in the population conforms to a more complex constrained syntactic structure (such as shown above in figure 1). The initial random population is created in accordance

with this constrained syntactic structure. Crossover must be performed in a structure-preserving way so as to preserve the syntactic validity of all offspring. In structure-preserving crossover, the architecture-defining points of an overall program are never eligible to be chosen as crossover points and are never altered by crossover. Instead, structure-preserving crossover is restricted to the work-performing points. In structure-preserving crossover, the work-performing points in the overall program are partitioned into a certain number of types.

The basic idea of structure-preserving crossover is that any work-performing point anywhere in the overall program is randomly chosen, without restriction, as the crossover point of the first parent. That point has a type assigned to it. Then, once the crossover point of the first parent has been chosen, the crossover point of the second parent is randomly chosen from among points of the same type.

The typing of the work-performing points of an overall program constrains the set of subtrees that can potentially replace the chosen crossover point and the subtree below it. This typing is done so that the structure-preserving crossover operation will always produce valid offspring.

There are several ways of assigning types to the work-performing points of an overall program.

(1) *Branch typing* assigns the same type to all the work-performing points of each separate branch of an overall program (but a different type to each different branch). There are as many types of work-performing points as there are branches in the overall program.

(2) *Like-Branch Typing* assigns the same type to all the work-performing points of each separate branch of an overall program and assigns a

different type to each different branch, except that if the function sets and terminal sets of two branches are identical, all the points of both such branches are assigned the same type.

(3)*Point typing* assigns a type to each individual work-performing point in the overall program reflective of both the branch where the point is located and the contents of the subtree starting at the point. The characteristics of the branch where the point is located is relevant in determining whether a subtree from another program may be inserted at the point. The contents of the subtree starting at the point are relevant in determining if the subtree may be inserted at a particular point of another program.

If a program is subject to any additional problem-specific constrained syntactic structure, that additional structure, if any, must also be considered in typing.

When all the programs in the population have a common architecture, any of the three methods of typing may be used. In practice, branch typing is most commonly used. The crossover operation starts with two parents and produces two offspring when either branch typing or like-branch typing is being used.

Point typing is used for architecturally diverse populations. If, for the sake of argument, branch typing or like-branch typing were to be used on an architecturally diverse population, the crossover operation would be virtually hamstrung; hardly any crossovers could occur. The types produced by branch typing or like-branch typing are insufficiently descriptive and overly constraining in an architecturally diverse population.

When point typing is used, the crossover operation acquires a directionality that did not exist with branch typing or like-branch typing. A distinction must be

made between the contributing (first) parent and the receiving (second) parent. Consequently, the crossover operation starts with two parents, but produces only one offspring.

The crossover point (called the *point of insertion*) of the receiving (second) parent must be chosen from the set of points for which the crossover fragment from the contributing (first) parent "has meaning" if the crossover fragment were to be inserted at the point.

When genetic material is inserted into the receiving parent during structure-preserving crossover with point typing, the offspring inherits its architecture from the receiving parent (the maternal line) and is guaranteed to be syntactically and semantically valid.

Point typing is governed by three general principles.

First, every terminal and function actually appearing in the crossover fragment from the contributing parent must be in the terminal set or function set of the branch of the receiving parent containing the point of insertion. This first general principle applies to actual variables of the problem, dummy variables, random constants, primitive functions, and automatically defined functions.

Second, the number of arguments of every function actually appearing in the crossover fragment from the contributing parent must equal the number of arguments specified for the same-named function in the argument map of the branch of the receiving parent containing the insertion point. This second general principle governing point typing applies to all functions. However, the emphasis is on the automatically defined functions because the same function name is used

to represent entirely different functions with differing number of arguments for different individuals in the population.

Third, all additional problem-specific syntactic rules of construction, if any, must be satisfied.

Structure-preserving crossover with point typing is described in detail in Koza 1994a.

Structure-preserving crossover with point typing permits robust recombination while simultaneously guaranteeing that any pair of architecturally different parents will produce syntactically and semantically valid offspring. In addition, structure-preserving crossover with point typing enables the architecture appropriate for solving the problem to be *evolutionarily selected* during a run while the problem is being solved. In addition, when the six new architecture-altering operations are being used, structure-preserving crossover with point typing enables the architecture appropriate for solving the problem to be *evolved* during a run while the problem is being solved in the sense of *actually changing* the architecture of programs dynamically during the run.

5.3. Steps for Executing Genetic Programming

Execution of genetic programming consists of the following steps. The six new architecture-altering operations appear as items (2)(c)(iii) through (2)(c)(ix).

The steps for executing genetic programming are as follows:

(1) Generate an initial random population (generation 0) of computer programs.

(2) Iteratively perform the following sub-steps until the termination criterion of the run has been satisfied:

(a) Execute each program in the population and assign it (explicitly or implicitly) a fitness value according to how well it solves the problem.

(b) Select program(s) from the population to participate in the genetic operations in (c) below.

(c) Create new program(s) for the population by applying the following genetic operations.

(i)*Reproduction*: Copy an existing program to the new population.

(ii)*Crossover*: Create new offspring program(s) for the new population by recombining randomly chosen parts of two existing programs.

(iii) *Mutation*. Create one new offspring program for the new population by randomly mutating a randomly chosen part of one existing program.

(iv) *Branch duplication*: Create one new offspring program for the new population by duplicating one function-defining branch of one existing program and making additional appropriate changes to reflect this change.

(v) *Argument duplication*: Create one new offspring program for the new population by duplicating one argument of one function-defining branch of one existing program and making additional appropriate changes to reflect this change.

(vi) *Branch deletion*: Create one new offspring program for the new population by deleting one function-defining branch of one

existing program and making additional appropriate changes to reflect this change.

(vii) *Argument deletion*: Create one new offspring program for the new population by deleting one argument of one function-defining branch of one existing program and making additional appropriate changes to reflect this change.

(viii) *Branch Creation*: Create one new offspring program for the new population by adding one new function-defining branch containing a portion of an existing branch and creating a reference to that new branch.

(ix) *Argument creation*: Create one new offspring program for the population by adding one new argument to the argument list of an existing function-defining branch and appropriately modifying references to that branch.

(3) After satisfaction of the termination criterion (which usually includes a maximum number of generations to be run as well as a problem-specific success predicate), the single best computer program in the population produced during the run (the best-so-far individual) is designated as the result of the run. This result may (or may not) be a solution (or approximate solution) to the problem.

6. Example of an Actual Run

The architecture-altering operations described herein will now be illustrated by showing an actual run of the problem of symbolic regression of the even-5-parity function. The Boolean even-*k*-parity function takes *k* Boolean arguments, `D0`, `D1`, `D2`, and so forth (up to a total of *k* arguments). The even-*k*-parity function

returns T (true) if an even number of its Boolean arguments are T, but otherwise returns NIL (false). Boolean parity functions are often used as benchmarks for experiments in machine learning because a change in any one input (environmental sensor) toggles the outcome. The problem is to discover a computer program that mimics the behavior of the Boolean even-*k*-parity problem for every one of the 2^k combinations of its *k* Boolean inputs. The primitive functions for this problem are AND, OR, NAND, and NOR.

6.1. Example with an Complete Genealogical Audit Trail

The run starts with the random creation of a population of 1,000 individual programs. The minimalist approach is used herein. That is, each program in the initial random population at generation 0 consists of one result-producing branch and one one-argument function-defining branch and has an argument map of {1}.

Thus, the terminal set for the result-producing branch, $\mathcal{T}_{rpb}$, for a program in the population for the Boolean even-3-parity problem is

$\mathcal{T}_{rpb}$ = {D0, D1, D2}.

The function set for the result-producing branch, $\mathcal{F}_{rpb}$, is

$\mathcal{F}_{rpb}$ = {AND, OR, NAND, NOR, ADF0},

with an argument map of

{2, 2, 2, 2, 1}.

The terminal set for the automatically defined function, ADF0, is

$\mathcal{T}_{adf0}$ = {ARG0}.

The function set, $\mathcal{F}_{adf0}$, for ADF0 is

$\mathcal{F}_{adf0}$ = {AND, OR, NAND, NOR},

with an argument map for this function set of

{2, 2, 2, 2}.

After creating the 1,000 programs for the initial random population, each program in the population is evaluated as to how well it solves the problem at hand. The fitness of a program in the population of 1,000 programs is measured according to how well that program mimics the target function for all eight combinations of three Boolean arguments. The raw fitness of a program is the number of matches.

In one particular run, the best program from among the 1,000 randomly created programs in generation 0 has the function-defining branch (defining ADF0) shown below:

```
(OR (AND (NAND ARG0 ARG0) (OR ARG0 ARG0)) (NOR (NOR ARG0 ARG0)
            (AND ARG0 ARG0))).
```

The behavior of this function-defining branch is the Boolean constant function zero (called "Always False").

The result-producing branch of this best-of-generation program from generation 0 ignores ADF0 and is shown below:

```
(NOR (AND D0(NOR D2 D1)) (AND (AND D2 D1))).
```

Of course, it should be no surprise that the function-defining branch of even the best program of the initial random generation is not particularly useful or that this branch is ignored by the result-producing branch. The minimalist approach is not intended to provide a highly useful function-defining branch, but rather merely to provide a starting point for the evolutionary process.

Table 5 shows the behavior of this program from generation 0. The first three columns show the values of the three Boolean variables, `D0`, `D1`, and `D2`. The fourth column shows the value produced by the overall program. The fifth column shows the value of the target function, the even-3-parity function. The last column shows how well the program performed at matching the behavior of the target function. As is shown, the program was correct for six of the eight possible combinations (fitness cases). Thus, the program scored a raw fitness of 6 (out of a possible 8).

A new population of 1,000 programs is then created from the existing population of 1,000 programs. Each successive generation of the population is created from the existing population by applying various genetic operations. Reproduction and crossover are the most frequently performed genetic operations. In addition, the architecture-altering operations described herein are used on this run. Mutation and other previously described genetic operations may also be used in the process (although they are not used here).

The raw fitness of the best-of-generation program for generation 5 improves to 7. That is, this program correctly mimics the behavior of the target even-3-parity function for seven of the eight fitness cases. The program achieving this new and higher level of fitness has a total of four branches (i.e., one result-producing

branch and three function-defining branches). The change in the number of branches from 1 at generation 0 to 4 at generation 5 is the consequence of the architecture-altering operations. In addition to its one result-producing branch, this best-of-generation program for generation 5 has branches defining ADF0 (taking two arguments), ADF1 (taking two arguments), and ADF2 (taking three arguments), so that its argument map is {2, 2, 3}. The result producing branch of this program is shown below:

```
(NOR (ADF2 D0 D2 D1) (AND (ADF1 D2 D1)D0)).
```

The first function-defining branch (defining ADF0) of the best-of-generation program for generation 5 takes two dummy arguments, ARG0 and ARG1, and is shown below. The existence of two dummy arguments in this function-defining branch is a consequence of an argument duplication operation. As it happens, the behavior of this ADF0 is not important because ADF0 is not referenced by the result-producing branch.

```
(OR (AND (NAND ARG0 ARG0) (OR ARG1 ARG0)) (NOR (NOR ARG1 ARG0)
            (AND ARG0 ARG1))).
```

The second function-defining branch (defining ADF1) of the best-of-generation program for generation 5 also takes two dummy arguments, ARG0 and ARG1, and is shown below. The existence of this second function-defining branch is a consequence of a branch duplication operation.

```
(OR (AND ARG0 ARG1) (NOR ARG0 ARG1)).
```

Table 6 shows the behavior of ADF1 of the best-of-generation program for generation 5 which, as can be seen, is equivalent to the even-2-parity function.

The function-defining branch for ADF2 of this best-of-generation program for generation 5 takes three dummy arguments, ARG0, ARG1, and ARG2, and is shown below. This third function-defining branch exists as a consequence of yet another branch duplication operation.

```
(AND ARG1 (NOR ARG0 ARG2)).
```

Table 7 shows that the behavior of ADF2 consists of returning 1 only when ARG0 and ARG2 are 0 and ARG1 is 1.

The raw fitness of the best individual program in the population remains at a value of 7 for generations 6, 7, 8, and 9; however, the average fitness of the population as a whole improves during these generations.

On generation 10, the best program in the population of 1,000 perfectly mimics the behavior of the even-3-parity function. This 100%-correct solution to the problem has a total of six branches (i.e., five function-defining branches and one result-producing branch). The argument map of this program is {2, 2, 3, 2, 2}. This multiplicity of branches is a consequence of the repeated application of the branch duplication operation and the branch creation operation. The function-defining branches of this program each have more than one dummy argument. All of these additional arguments exist as a consequence of the repeated application of the argument duplication operation.

The result-producing branch of this best-of-generation program for generation 10 is shown below:

```
(NOR (ADF4 D0(ADF1 D2 D1)) (AND (ADF1 D2 D1) D0))
```

The function-defining branch for ADF0 of this best-of-generation program for generation 10 takes two dummy arguments, ARG0 and ARG1, and is shown below. The behavior of ADF0 is equivalent to the odd-2-parity function.

```
(OR (AND (NAND ARG0 ARG0) (OR ARG1 ARG0)) (NOR (NOR ARG1 ARG0)
            (AND ARG0 ARG1)))
```

The function-defining branch for ADF1 of the best-of-generation program for generation 10 takes two dummy arguments, ARG0 and ARG1, and is shown below. ADF1 is equivalent to the even-2-parity function.

```
(OR (AND ARG0 ARG1) (NOR ARG0  ARG1))
```

The function-defining branch for ADF2 takes three dummy arguments, ARG0, ARG1, and ARG2, and is shown below. ADF2 returns 1 only when ARG0 and ARG2 are 0 and ARG1 is 1. However, ADF2 is ignored by the result-producing branch.

```
(AND ARG1 (NOR ARG0 ARG2))
```

The function-defining branch for ADF3 is the one-argument identity function. This relatively useless branch is ignored by the result-producing branch.

The function-defining branch for ADF4 of the best-of-generation program for generation 10 takes two dummy arguments, ARG0 and ARG1, and is shown below. ADF4 is equivalent to the even-2-parity function.

```
(OR (AND ARG0 ARG1) (NOR ARG0  ARG1))
```

Since both ADF1 and ADF4 are both even-2-parity functions, the result-producing branch can be simplified to the expression below. This expression is equivalent to the even-3-parity function.

```
(NOR (EVEN-2-PARITY D0(EVEN-2-PARITY D2 D1)) (AND (EVEN-2-PARITY
            D2 D1) D0))
```

An examination of the genealogical audit trail shows the interplay between the Darwinian reproduction operation, the one-offspring crossover operation using point typing, and the new architecture-altering operations.

Figure 3 shows all of the ancestors of the just-described 100%-correct solution from generation 10 of the run in example 1 of the problem of symbolic regression of the even-3-parity problem. The generation numbers (from 0 to 10) are shown on the left edge of figure 3. Figure 3 also shows the sequence of reproduction operations, crossover operations, and architecture-altering operations that gave rise to every program that was an ancestor to the 100%-correct program in generation 10. The 100%-correct solution from generation 10 is represented by the box labeled M10 at the bottom of the figure. The argument map of this solution, namely {2, 2, 3, 2, 2}, is shown in this box.

The two lines flowing into the box M10 indicate that the solution in generation 10 was produced by a crossover operation acting on two programs from the previous generation (generation 9). Figure 3 uses the convention of placing the mother M9 (the receiving parent) on the right and father P9 (the contributing parent) on the left. Recall that, in a one-offspring crossover operation using point

typing, the bulk of the structure of a multi-part program comes from the mother since the father contributes only one subtree into only one of the many branches of the mother. Thus, the 11 boxes on the right side of this figure (consecutively numbered from M0 to M10) represent the maternal genetic lineage (from generations 0 through generation 10) of the 100%-correct solution M10 that emerged in generation 10. The 100%-correct solution M10 in generation 10 has the same argument map, {2, 2, 3, 2, 2}, as the mother M9 because the crossover operation is not an architecture-altering operation and does not change the architecture (or argument map) of the offspring (relative to the mother).

The maternal lineage will now be reviewed in detail so as to illustrate the overall process of evolving the architecture of a solution to a problem while simultaneously evolving the solution to the problem.

The mother M9 from generation 9 (shown on the right side of figure 3) has an argument map of {2, 2, 3, 2, 2}, has a raw fitness of 7, was itself the result of a crossover of two parents from generation 8. The grandfather of the 100%-correct solution M10 in generation 10 (and the father of M9) was P8. The grandmother of the 100%-correct solution M10 in generation 10 (and the mother of M9) was M8.

The grandmother M8 from generation 8 of the 100%-correct solution M10 in generation 10 (and the mother of M9) has an argument map of {2, 2, 3, 2, 2}, has a raw fitness of 7, and was the result of a branch duplication from a single ancestor M7 from generation 7.

Because of the branch duplication operation, the program M7 from generation 7 of the maternal lineage at the far right of figure 3 has one fewer branch than its

offspring M8. Program M7 has an argument map of {2, 2, 3, 2}. Program M7 was the result of an argument duplication from a single ancestor from generation 6.

Because of the argument duplication operation, the fourth function-defining branch of the program M6 from generation 6 of the maternal lineage at the far right of figure 3 has one less argument than its offspring M7. Program M6 from generation 6 has an argument map of {2, 2, 3, 1} whereas program M7 from generation 7 has an argument map of {2, 2, 3, 2}. Program M6 was the result of an branch creation from a single ancestor M5 from generation 5.

Because of the branch creation operation, the program M5 from generation 5 shown on the right side of figure 3 has one fewer function-defining branch then program M6. Program M5 has an argument map of {2, 2, 3}. In turn, program M5 was the result of a branch creation from a single ancestor M4 from generation 4.

Program M4 from generation 4 shown on the right side of figure 3 has one less function-defining branch than its offspring program M5. Program M4 has an argument map of {2, 2}. Program M4 was the result of a reproduction operation from a single ancestor M3 from generation 3.

Program M4 from generation 3 shown on the right side of figure 3 has an argument map of {2, 2} and was the result of a crossover involving father P2 and mother M2 from generation 2.

Program M2 from generation 2 (shown on the right side of figure 3) has an argument map of {2, 2} and was the result of a branch creation from a single ancestor M1 from generation 1.

Program M1 from generation 1 has an argument map of {2} and was the result of an argument duplication of a single ancestor M0 from generation 0.

Program `M0` from generation 0 at the upper right corner of figure 3 has an argument map of {1} and has a raw fitness of 6. It has an argument map of {1} because all programs at generation 0 consist of one result-producing branch and a single one-argument function-defining branch when the minimalist approach is being used.

The sequence of genetic operations and architecture-altering operations of this run shows the simultaneous evolution of the architecture while solving the problem.

6.2. Example with Even-5-Parity Problem

For the problem of symbolic regression of the even-5-parity function, A population size, *M*, of 96,000 was used. The targeted maximum number of generations, *G*, was set at 76. The run uses the "minimalist approach" in which each program in generation 0 consists of one result-producing branch and a single two-argument function-defining branch. Branch deletion and argument deletion with random regeneration were used. The percentage of operations at each processing node on each generation was 74% crossovers; 10% reproductions; 0% mutations; 5% branch duplications, 5 argument duplications; 0.5% branch deletions; 0.5% argument deletions; 5% branch creations; and 0% argument creations. Other minor parameters were chosen as in Koza 1994a.

The problem was run on a home-built medium-grained parallel computer system. In the so-called *distributed genetic algorithm* or *island model* for parallelization (Tanese 1989), different semi-isolated subpopulations (called *demes* after Sewall Wright 1943) are situated at the different processing nodes of the system. The system consisted of a host PC 486 type computer running Windows and 64 Transtech TRAMs (containing one INMOS T805 transputer and 4 megabytes of RAM memory) arranged in a toroidal mesh. There were $D = 64$

demes, a population size of Q = 1,500 per deme, and a migration rate (boatload size) of B = 8% (in each of four directions on each generation for each deme). Generations are run asynchronously. Additional details of the parallel implementation of genetic programming on a network of transputers can be found in Koza and Andre 1995.

On generation 13 of one run, a 100%-correct solution to the even-5-parity problem emerged in the form of a computer program with one three-argument automatically defined function and one two-argument automatically defined function.

Three-argument ADF0 (which had only two arguments in generation 0) performs Boolean rule 106, a non-parity rule, and is below:

```
(NOR (OR (AND (OR (OR ARG0 ARG2) (NAND ARG0 ARG2)) (AND (NAND ARG0
ARG1) (NOR ARG2 ARG0))) (AND (AND (NOR ARG1 ARG0) (OR ARG2 ARG0))
(OR (NAND ARG0 ARG1) (NOR ARG2 ARG0)))) (NAND (NAND (AND (NOR ARG0
ARG2) (NAND ARG0 ARG0)) (NOR (NAND ARG0 ARG0) (NOR ARG2 ARG1)))
(OR (NAND (AND ARG1 ARG0) (OR ARG1 ARG0)) (OR (NAND ARG2 ARG2)
(NOR ARG0 ARG0)))))
```

Two-argument ADF1 (which did not exist at all in generation 0) is equivalent to the odd-2-parity function and is below:

```
(NOR (OR (AND (OR (OR ARG0 ARG1) (NAND ARG0 ARG1)) (AND (NAND ARG0
ARG1) (NOR ARG1 ARG0))) (AND (AND (NOR ARG1 ARG0) (OR ARG1 ARG0))
(OR (NAND ARG0 ARG1) (NOR ARG1 ARG0)))) (NAND (NAND (AND (NOR ARG0
ARG1) (NAND ARG0 ARG0)) (NOR (NAND ARG0 ARG0) (NOR ARG1 ARG1)))
```

```
(OR (NAND (AND ARG1 ARG0) (OR ARG1 ARG0)) (OR (NAND ARG1 ARG1)
(NOR ARG0 ARG0))))).
```

The result-producing branch of this program invokes both `ADF0` and `ADF1` and is below:

```
(AND (OR (ADF0 (NAND D1 D2) (ADF0 D2 D0 D0) (ADF0 D2 D0 D0)) (NAND
(OR D3 D1) (ADF1 D3 D3))) (ADF0 (ADF1 (NAND D1 D2) (NOR D4 D4))
(ADF1 (ADF1 D3 D0) (NOR D1 D2)) (ADF1 (ADF1 D3 D0) (NOR D1 D2))))
```

7. Performance Characteristics of the New Operations

We now use the Boolean even-5-parity problem to compare, over a series of runs, three performance characteristics of the architecture-altering operations for the following five approaches:

(A) **without automatically defined functions** (corresponding to the style of runs discussed throughout most of *Genetic Programming*),

(B) with automatically defined functions, **evolutionary selection** of the architecture (corresponding to the style of runs discussed in chapters 21–25 of *Genetic Programming II* on the evolutionary selection of the architecture), an architecturally diverse initial population, and structure-preserving crossover with point typing,

(C) with automatically defined functions, the **architecture-altering operations**, an architecturally diverse population (after generation 0), and structure-preserving crossover with point typing,

(D) with automatically defined functions, a fixed, user-supplied architecture (i.e., an argument map of {3, 2} that is known to be a good choice of architecture for this problem), and structure-preserving crossover with **point typing**, and

(E) with automatically defined functions, the fixed, known-good, user-supplied {3, 2} architecture, and structure-preserving crossover with **branch typing** (corresponding to the style of runs discussed throughout most of *Genetic Programming II*).

The comparisons are made for the following three performance characteristics: computational effort, E (with 99% probability); the wallclock time, $W(M,t,z)$ in seconds(with 99% probability); and the average structural complexity, $\bar{S}$. These three measures are described in detail in Koza 1994a.

The comparisons in table 8 all used a common population size, M, of 96,000. All runs solved well before the targeted maximum number of generations, G, of 76.

As can be seen from the table, all four approaches (B, C, D, or E) employing automatically defined functions require less computational effort than not using them (approach A). Approach E (which benefits from the most user-supplied information) requires the least computational effort. At the other extreme, approach A requires the most computational effort.

Approach C (using the architecture-altering operations) requires less computational effort than solving the problem without automatically defined functions (approach A), but more computational effort than with the fixed, known-good, user-supplied architecture (approach E).

Approach B requires greater computational effort than approach C, but less than that for approach A.

Approach D isolates the additional computational effort required by point typing (relative to approach E). Greater computational effort is required by

approach D than approach E. Since the computational effort for approach C is virtually tied with approach D, the cost of architecture-altering operations for this problem is not much greater than the cost of point typing.

Approach E consumes less wallclock time than approach C (using the architecture-altering operations), which, in turn, consumes less wallclock time than approach A (without automatically defined functions).

The average structural complexity, $\bar{S}$, for all four approaches (B, C, D, or E) employing automatically defined functions is less than that for approach A (without automatically defined functions). Approach C (using the architecture-altering operations) has the lowest value of $\bar{S}$.

Note also that all four approaches (B, C, D, or E) employing ADFs require less computational effort, require less wallclock time, and produce smaller solutions (i.e., are more parsimonious) than the ADF-less approach (approach A).

Additional work in this area is described in Koza (1995a, b, c) and Koza and Andre (1996). Application of the architecture-altering operations to the automated design of analog electrical circuits using genetic programming with automatically defined functions is described in Koza, Andre, Bennett, and Keane (1996).

8. Conclusions

This chapter describes how the biological theory of gene duplication described in Susumu Ohno's provocative book, *Evolution by Means of Gene Duplication*, was brought to bear on a vexatious problem of architecture discovery for automated machine learning. The resulting biologically-motivated approach enables genetic programming to automatically discover the size and shape of the solution at the same time as genetic programming is evolving a solution to the problem. This is accomplished using six biologically-motivated architecture-altering operations

that provide a way to automatically discover, during a run of genetic programming, both the architecture and the sequence of steps of a multi-part computer program that will solve the given problem.

9. Acknowledgements

David Andre and Walter Alden Tackett wrote the program in C for the architecture-altering operations used above. The midge comes from "Destructive and useful insects" by C.L. Metcalf and W.P Flint.

10. Bibliography

Angeline, Peter J. and Kinnear, Kenneth E. Jr. (editors). 1996. *Advances in Genetic Programming 2*. Cambridge, MA: The MIT Press.

Angeline, Peter J. and Pollack, Jordan B. Coevolving high-level representations. In Langton, Christopher G. (editor). *Artificial Life III, SFI Studies in the Sciences of Complexity*. Volume XVII Redwood City, CA: Addison-Wesley. Pages 55–71. 1994**.**

Brooks Low, K. 1988. Genetic recombination: A brief overview. In Brooks Low, K. (editor) *The Recombination of Genetic Material*. San Diego: Academic Press. Pages 1–21.

Cavicchio, Daniel J. 1970. *Adaptive Search using Simulated Evolution*. Ph.D. dissertation. Department of Computer and Communications Science, University of Michigan.

Darwin, Charles. 1859. *On the Origin of Species by Means of Natural Selection.* John Murray.

Dyson, P. and Sherratt, D. 1985. Molecular mechanisms of duplication, deletion, and transposition of DNA. In Cavalier-Smith, T. (editor). *The Evolution of Genome Size*. Chichester: John Wiley & Sons.

Galli, Joakim and Wislander, Lars. 1993. Two secretary protein genes in *Chironomus tentans* have arisen by gene duplication and exhibit different developmental expression patterns. *Journal of Molecular Biology*. 231:324–334.

Galli, Joakim and Wislander, Lars. 1994. Structure of the smallest salivary-gland secretory protein in *Chironomus tentans*. *Journal of Molecular Evolution*. 38:482-488.

Go, Mittko. 1991. Module organization in proteins and exon shuffling. In Osawa, S. and Honjo, T. (editors). *Evolution of Life*. Tokyo: Springer-Verlag.

Goldberg, David E. 1989. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley.

Goldberg, David E., Korb, Bradley, and Deb, Kalyanmoy. 1989. Messy genetic algorithms: Motivation, analysis, and first results. *Complex Systems*. 3(5): 493–530.

Gruau, Frederic. 1994. Genetic micro programming of neural networks. In Kinnear, Kenneth E. Jr. (editor). *Advances in Genetic Programming*. Cambridge, MA: The MIT Press. Pages 495–518.

Holland, John H. 1975. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. Ann Arbor, MI: University of Michigan Press. The second edition is currently available from The MIT Press 1992.

Hood, Leory and Hunkapiller, Tim. 1991. Modular evolution and the immunoglobin gene superfamily. In Osawa, S. and Honjo, T. (editors). *Evolution of Life*. Tokyo: Springer-Verlag.

Kinnear, Kenneth E. Jr. (editor). 1994. *Advances in Genetic Programming*. Cambridge, MA: The MIT Press.

Koza, John R. 1989. Hierarchical genetic algorithms operating on populations of computer programs. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*. San Mateo, CA: Morgan Kaufmann. I:768-774.

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

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

Koza, John R. 1994b. *Genetic Programming II Videotape: The Next Generation*. Cambridge, MA: The MIT Press.

Koza, John R. 1994c. *Architecture-altering operations for evolving the architecture of a multi-part program in genetic programming*. Stanford University Computer Science Department technical report STAN-CS-TR-94-1528. October 21, 1994.

Koza, John R. 1995a. Evolving the architecture of a multi-part program in genetic programming using architecture-altering operations. In McDonnell, John R., Reynolds, Robert G., and Fogel, David B. (editors). 1995. *Evolutionary Programming IV: Proceedings of the Fourth Annual Conference on Evolutionary Programming*. Cambridge, MA: The MIT Press. Pages 695–717.

Koza, John R. 1995b. Gene duplication to enable genetic programming to concurrently evolve both the architecture and work-performing steps of a

computer program. *Proceedings of the 14th International Joint Conference on Artificial Intelligence*. San Francisco, CA: Morgan Kaufmann. Pages 734–740.

Koza, John R. 1995c. Two ways of discovering the size and shape of a computer program to solve a problem. In Eshelman, Larry J. (editor). Proceedings of the Sixth International Conference on Genetic Algorithms. San Francisco, CA: Morgan Kaufmann Publishers. Pages 287–294.

Koza, John R. and Andre, David. 1995. *Parallel Genetic Programming on a Network of Transputers*. Stanford University Computer Science Department technical report STAN-CS-TR-95-1542. January 30, 1995.

Koza, John R. and Andre, David. 1996. Classifying protein segments as transmembrane domains using architecture-altering operations in genetic programming. In Angeline, Peter J. and Kinnear, Kenneth E. Jr. (editors). 1996. *Advances in Genetic Programming 2*. Cambridge, MA: The MIT Press.

Koza, John R., Andre, David, Bennett III, Forrest H, and Keane, Martin A. 1996. Use of automatically defined functions and architecture-altering operations in automated circuit synthesis with genetic programming. In Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors). 1996. *Genetic Programming 1996: Proceedings of the First Annual Conference, July 28-31, 1996, Stanford University*. Cambridge, MA: The MIT Press. Pages 132–140.

Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors). 1996. *Genetic Programming 1996: Proceedings of the First Annual Conference, July 28-31, 1996, Stanford University*. Cambridge, MA: The MIT Press.

Koza, John R., and Rice, James P. 1992. *Genetic Programming: The Movie*. Cambridge, MA: The MIT Press.

Lazcano, A. and Miller, S. L. 1994. How long did it take for life to begin and evolve to cyanobacteria? *Journal of Molecular Evolution*. 39:546–554.

Lindgren, Kristian. 1991. Evolutionary phenomena in simple dynamics. In Langton, Christopher, Taylor, Charles, Farmer, J. Doyne, and Rasmussen, Steen (editors). *Artificial Life II, SFI Studies in the Sciences of Complexity*. Volume X. Redwood City, CA: Addison-Wesley. Pages 295-312.

Maeda, Nobuyo and Smithies, Oliver. 1986. The evolution of multigene families: Human haptoglobin genes. *Annual Review of Genetics*. 20:81-108.

Mitchell, Melanie. 1996. *An Introduction to Genetic Algorithms*. Cambridge, MA: The MIT Press.

Nei, Masatoshi. 1987. *Molecular Evolutionary Genetics*. New York: Columbia University Press.

Ohno, Susumu. 1970. *Evolution by Gene Duplication*. New York: Springer-Verlag.

Patthy, Laszlo. 1991. Modular exchange principles in proteins. *Current Opinion in Structural Biology*. 1:351–361.

Riley, M. 1993. Functions of the gene products of *Escherichia coli*. *Reviews of Microbiology*. 32:519–560.

Samuel, Arthur L. 1959. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development.* 3(3): 210–229.

Smith, T. F. and Waterman, M. S. 1981.Identification of common molecular subsequences. *Journal of Molecular Biology*. 147:195-197.

Stryer, Lubert. 1988. *Biochemistry*. W. H. Freeman. Third Edition.

Tanese, Reiko. 1989. *Distributed Genetic Algorithm for Function Optimization.* PhD. dissertation. Department of Electrical Engineering and Computer Science. University of Michigan.

Wright, Sewall. 1943. Isolation by distance. *Genetics*. 28:114–138.

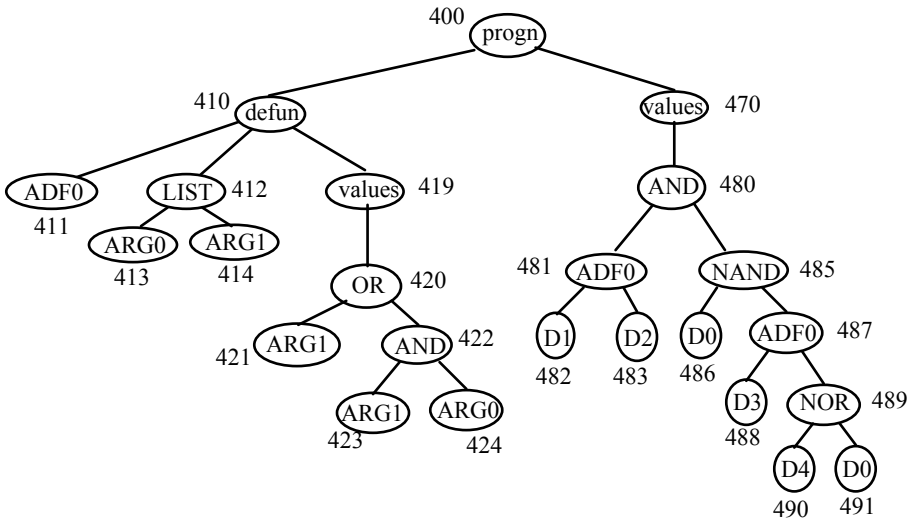

Figure 1 Program with an argument map of {2} consisting of one two-argument function-defining branch (`ADF0`) and one result-producing branch.

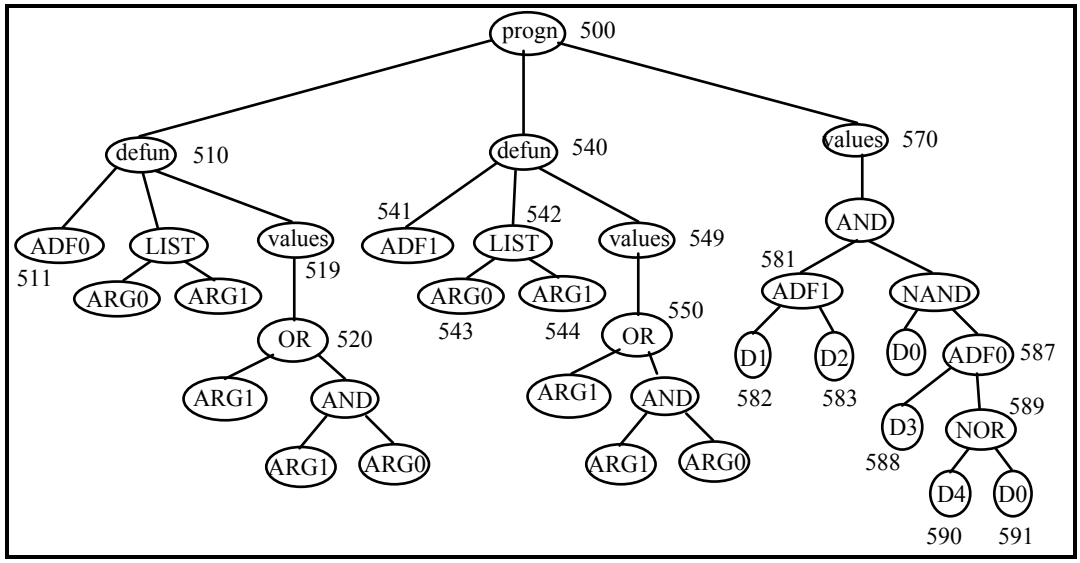

Figure 2 Program with an argument map of {2, 2} consisting of two two-argument function-defining branches (`ADF0` and `ADF1`) and one result-producing branch.

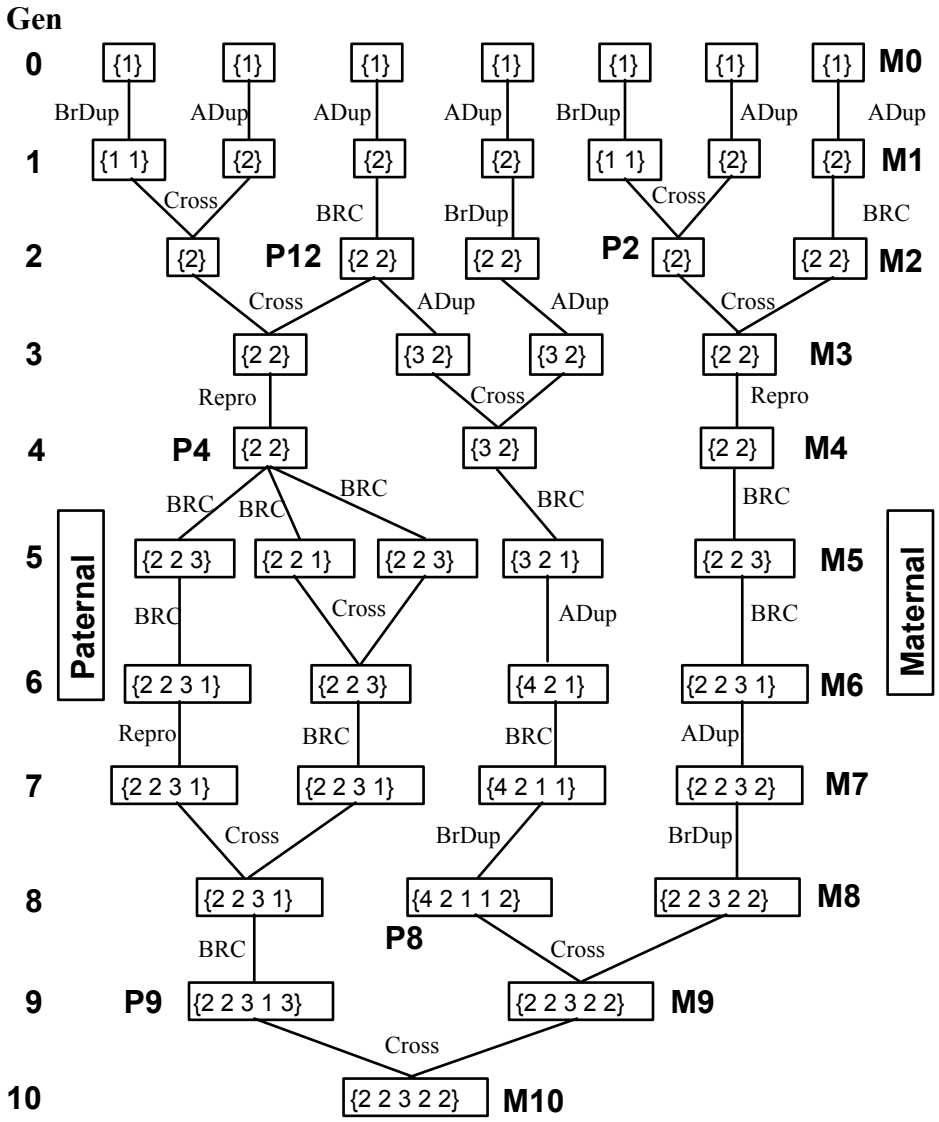

Figure 3 Complete genealogical audit trail showing all of the ancestors in generations 0 through 9 for the ultimate solution M10 from generation 10 for

the run in example 1. The maternal line is shown at the right and the paternal line at the left.

Table 1 Portion of a DNA sequence containing the two expressed proteins.

```
TGAAGTAATA   TTAAGCTATG   AGAATTAAGT   TCCTAGTAGT   ATTAGCAGTT   950
                   M      R  I  K  F     L  V  V     L  A  V

ATCTGCTTGT   TTGCACATTA   TGCCTCAGCT   AGTGGTATGG   GGGGTGATAA   1000
I  C  L  F     A  H  Y     A  S  A     S  G  M  G     G  D  K

AAAACCCAAA   GATGCCCCAA   AACCCAAAGA   TGCCCCAAAA   CCCAAAGAAG   1050
 K  P  K     D  A  P  K     P  K  D     A  P  K     P  K  E  V

TGAAGCCTGT   CAAAGCTGAG   TCATCAGAGT   ATGAGATAGA   AGTCATTAAA   1100
  K  P  V     K  A  E     S  S  E  Y     E  I  E     V  I  K

CACCAGAAAG   AAAAGACCGA   GAAGAAGGAG   AAGGAGAAGA   AGACTCACGT   1150
H  Q  K  E     K  T  E     K  K  E     K  E  K  K     T  H  V

TGAAACCAAG   AAAGAAGTTA   AAAAGAAGGA   GAAGAAGCAA   ATCCCTTGTT   1200
 E  T  K     K  E  V  K     K  K  E     K  K  Q     I  P  C  S

CTGAAAAACT   CAAGGATGAA   AAACTTGATT   GTGAGACCAA   GGGCGTCCCT   1250
  E  K  L     K  D  E     K  L  D  C     E  T  K     G  V  P

GCAGGCTACA   AAGCAATCTT   CAAATTCACA   GAAAACGAGG   AGTGCGATTG   1300
A  G  Y  K     A  I  F     K  F  T     E  N  E  E     C  D  W

GACGTGCGAT   TATGAAGCAC   TTCCACCACC   TCCAGGAGCA   AAGAAAGACG   1350
 T  C  D     Y  E  A  L     P  P  P     P  G  A     K  K  D  D

ACAAGAAAGA   AAAGAAGACA   GTTAAAGTCG   TTAAGCCACC   AAAGGAGAAA   1400
  K  K  E     K  K  T     V  K  V  V     K  P  P     K  E  K

CCACCAAAGA   AGCTTAGAAA   GGAATGCTCT   GGCGAAAAAG   TGATCAAATT   1450
P  P  K  K     L  R  K     E  C  S     G  E  K  V     I  K  F

CCAAAACTGT   CTCGTTAAGA   TTAGAGGACT   TATTGCCTTT   GGTGATAAGA   1500
 Q  N  C     L  V  K  I     R  G  L     I  A  F     G  D  K  T

CAAAGAACTT   TGATAAGAAG   TTCGCAAAGC   TTGTCCAAGG   AAAGCAGAAG   1550
  K  N  F     D  K  K     F  A  K  L     V  Q  G     K  Q  K

AAGGGCGCAA   AAAAAGCTAA   AGGCGGTAAG   AAGGCAGCAC   CAAAACCAGG   1600
K  G  A  K     K  A  K     G  G  K     K  A  A  P     K  P  G

ACCAAAACCA   GGGCCAAAAC   AAGCTGATAA   ACCAAAAGAT   GCAAAAAAAT   1650
 P  K  P     G  P  K  Q     A  D  K     P  K  D     A  K  K

AAACTGACAT   AGTAAGAATA   ATAAAATAAA   CATTATTTGA   GCAACATCAC   1700
AACACAAGAA   AAAAATCATA   TCAACATAAT   TAAGACCTAA   AAATTCTCGC   1750
TATTCACTTT   TTTTCAAATG   AATATCCAAA   ACAACATCAT   TAAGGGATCT   1800
TACACAATTT   TATCCCAAAT   TAGTTTTAAG   TCTATTTTTT   AGTTTTAAGT   1850
AAAACATTAG   TTAGAGAAAT   TTCAAATGCG   AAAAAAAGAC   AAAATCAAAA   1900
TTAACTCCAA   CTAATTGTCT   AGATCTAATC   ACCACTGAAA   AACAATATTT   1950
TTTTCAATAA   TATCTGAGAT   GAAAATTTTG   TAAGATACGA   TTCAAAAAAA   2000
AAAAAACAAA   AACTTAAATA   TTTTCTTTAT   AAGAAAGTAA   AAAACTTACA   2050
TGAACAACAA   GTAGACTAAG   GGCTTAAAAA   TACTAAGGAA   TTTAAAGAAA   2100
```

```
CTGAACCAAT   AACATCCAAT   AAATATAAGC   GTGTATTTAA   CATCCATTCA   2150
TGCAAAATTT   GACTTGTTTT   ATTCTAAACT   TTTGAATTGT   GAATATTTTT   2200
GATGATTATT   GAATATTTTA   CAGCATTTTT   CGACAAAATC   CAAGGAAACT   2250
GTTTTGTTTA   ATATATACTA   CAGCTCAGTA   TCTATGCACA   CGAAAAACTG   2300
TAACAGACCA   GACCATAAAA   CCTACACATC   ACCAAGATAC   GTATTTTAAA   2350
TTCATGTGAC   TGACAAAAGC   TGGAAACACT   TGTGTCACGT   CATGAAAACC   2400
TCGTTGAAAT   AAAACTTCTA   GAAAGGTTAT   CATGAAAGAG   TATAAAAGAG   2450
ATCTCAAACG   AGGCTCAGTC   AGTTCAGTTT   AGCTTGGACT   TCATATGAAG   2500
TAATATTTAG   CTATGAGAAT   TAAGTTCCTA   GTAGTATTAG   CAGTTATCTG   2550
               M  R  I      K  F  L      V  V  L  A      V  I  C

CTTGCTTGCA   CATTATGCCT   CAGCTAGTGG   TATGGGGGGT   GATAAAAAAC   2600
 L  L  A      H  Y  A  S      A  S  G      M  G  G      D  K  K  P

CCAAAGATGC   CCCAAAACCC   AAAGATGCCC   CAAAACCCAA   AGAAGTGAAG   2650
  K  D  A      P  K  P      K  D  A  P      K  P  K      E  V  K

CCTGTCAAAG   CTGACTCATC   AGAGTATGAG   ATAGAAGTCA   TTAAACACCA   2700
P  V  K  A      D  S  S      E  Y  E      I  E  V  I      K  H  Q

GAAAGAAAAG   ACCGAGAAGA   AGGAGAAGGA   GAAGAAAGCT   CACGTCGAAA   2750
 K  E  K      T  E  K  K      E  K  E      K  K  A      H  V  E  I

TCAAGAAAAA   GATTAAAAAT   AAGGAGAAGA   AGTTTGTCCC   ATGTTCTGAA   2800
  K  K  K      I  K  N      K  E  K  K      F  V  P      C  S  E

ATTCTCAAGG   ATGAAAAACT   TGAATGTGAG   AAAAATGCTA   CTCCAGGCTA   2850
I  L  K  D      E  K  L      E  C  E      K  N  A  T      P  G  Y

TAAAGCACTC   TTCGAATTCA   AAGAAAGCGA   AAGTTTTTGC   GAATGGGAGT   2900
 K  A  L      F  E  F  K      E  S  E      S  F  C      E  W  E  C

GCGATTATGA   AGCAATTCCA   GGAGCAAAGA   AAGACGAAAA   AAAGGAGAAG   2950
  D  Y  E      A  I  P      G  A  K  K      D  E  K      K  E  K

AAGGTAGTTA   AAGTCATTAA   GCCACCAAAG   GAAAAACCAC   CAAAGAAGCC   3000
K  V  V  K      V  I  K      P  P  K      E  K  P  P      K  K  P

TAGAAAGGAA   TGCTCTGGCG   AAAAAGTGAT   CAAATTCCAA   AACTGTCTCG   3050
 R  K  E      C  S  G  E      K  V  I      K  F  Q      N  C  L  V

TTAAGATTAG   AGGACTTATT   GCCTTTGGTG   ATAAGACAAA   GAACTTTGAT   3100
  K  I  R      G  L  I      A  F  G  D      K  T  K      N  F  D

AAGAAGTTTG   CAAAGCTTGT   CCAAGGAAAG   CAAAAGAAGG   GCGCAAAAAA   3150
K  K  F  A      K  L  V      Q  G  K      Q  K  K  G      A  K  K

AGCTAAAGGC   GGTAAGAAGG   CAGAACCAAA   ACCAGGACCA   AAACCAGCAC   3200
 A  K  G      G  K  K  A      E  P  K      P  G  P      K  P  A  P

CAAAACCAGG   ACCAAAACCA   GCACCAAAAC   CAGTACCAAA   ACCAGCTGAT   3250
  K  P  G      P  K  P      A  P  K  P      V  P  K      P  A  D

AAACCAAAAG   ATGCAAAAAA   ATAAACTGAC   ATAGTGAGAA   TAATAAAATA   3300
K  P  K  D      A  K  K
```

Table 2 Protein sequence of "A" protein.

MRIKFLVVLA	VICLFAHYAS	ASGMGGDKKP	KDAPKPKDAP	KPKEVKPVKA	50
ESSEYEIEVI	KHQKEKTEKK	EKEKKTHVET	KKEVKKKEKK	QIPCSEKLKD	100
EKLDCETKGV	PAGYKAIFKF	TENEECDWTC	DYEALPPPPG	AKKDDKKEKK	150
TVKVVKPPKE	KPPKKLRKEC	SGEKVIKFQN	CLVKIRGLIA	FGDKTKNFDK	200
KFAKLVQGKQ	KKGAKKAKGG	KKAAPKPGPK	PGPKQADKPK	DAKK	244

Table 3 Protein sequence of "B" protein.

MRIKFLVVLA	VICLLAHYAS	ASGMGGDKKP	KDAPKPKDAP	KPKEVKPVKA	50
DSSEYEIEVI	KHQKEKTEKK	EKEKKAHVEI	KKKIKNKEKK	FVPCSEILKD	100
EKLECEKNAT	PGYKALFEFK	ESESFCEWEC	DYEAIPGAKK	DEKKEKKVVK	150
VIKPPKEKPP	KKPRKECSGE	KVIKFQNCLV	KIRGLIAFGD	KTKNFDKKFA	200
KLVQGKQKKG	AKKAKGGKKA	EPKPGPKPAP	KPGPKPAPKP	VPKPADKPKD	250
AKK					253

Table 4 Protein alignment of the "A" and "B" proteins.

```
First.protein     MRIKFLVVLA VICLFAHYAS ASGMGGDKKP KDAPKPKDAP KPKEVKPVKA     50
Second.protein    MRIKFLVVLA VICLLAHYAS ASGMGGDKKP KDAPKPKDAP KPKEVKPVKA     50

First.protein     ESSEYEIEVI KHQKEKTEKK EKEKKTHVET KKEVKKKEKK QIPCSEKLKD    100
Second.protein    DSSEYEIEVI KHQKEKTEKK EKEKKAHVEI KKKIKNKEKK FVPCSEILKD    100

First.protein     EKLDCETKGV PAGYKAIFKF TENEE-CDWT CDYEALPPPP GAKKDDKKEK    149
Second.protein    EKLECEKNAT P-GYKALFEF KESESFCEWE CDYEAI---P GAKKDEKKEK    146

First.protein     KTVKVMKPPK EKPPKKLRKE CSGEKVIKFQ NCLVKIRGLI AFGDKTKNFD    199
Second.protein    KMVKVIKPPK EKPPKKPRKE CSGEKVIKFQ NCLVKIRGLI AFGDKTKNFD    196

First.protein     KKFAKLVQGK QKKGAKKAKG GKKAAPKPGP KPGPK----Q ADKP------    239
Second.protein    KKFAKLVQGK QKKGAKKAKG GKKAEPKPGP KPAPKPGPKP APKPVPKPAD    246

First.protein     --KDAKK                                                   244
Second.protein    KPKDAKK                                                   253
```

Table 5 Operation of the best-of-generation program from generation 0.

D0	D1	D2	BEST-OF-GENERATION PROGRAM FOR GENERATION 0	EVEN-3-PARITY FUNCTION	SCORE
0	0	0	1	1	correct
0	0	1	1	0	wrong
0	1	0	1	0	wrong
0	1	1	1	1	correct
1	0	0	0	0	correct
1	0	1	1	1	correct
1	1	0	1	1	correct
1	1	1	0	0	correct

Table 6 ADF1 of the best-of-generation program of generation 5.

ARG0	ARG1	ADF0
0	0	1
0	1	0
1	0	0
1	1	1

Table 7 ADF2 of the best-of-generation program of generation 5.

ARG0	ARG1	ARG2	ADF2
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

Table 8 Comparison of the five approaches.

Approach	Runs	Computational effort E	Wallclock time $W(M,t,z)$	Average Size of solution $\bar{S}$
A - No ADFs	14	5,025,000	36,950	469.1
B - ADFs + Evolutionary Selection of Architecture	14	4,263,000	66,667	180.9
C - ADFs + Architecture-Altering Operations	25	1,789,500	13,594	88.8
D - ADFs + Point Typing–Fixed, Known-Good Architecture	25	1,705,500	14,088	130.0
E - ADFs + Branch Typing–Fixed, Known-Good Architecture	25	1,261,500	6,481	112.2