Evolving the Architecture of a Multi-Part Program in Genetic Programming Using Architecture-Altering Operations

John R. Koza

Abstract: This paper describes six new architecture-altering operations that provide a way to dynamically determine the architecture of a multi-part program during a run of genetic programming. The new operations are patterned after the naturally occurring operations of gene duplication and gene deletion and are motivated by Ohno's provocative book *Evolution by Means of Gene Duplication*. The new operations are branch duplication, argument duplication, branch creation, argument creation, branch deletion, and argument deletion. These operations dynamically change the architecture of various programs during a run of genetic programming. The new operations can also be interpreted as providing an automated way to specialize and generalize programs. The paper demonstrates that problems can be solved while the architecture is being evolved.

1 INTRODUCTION

In nature, sexual recombination ordinarily recombines a part of the chromosome of one parent with a corresponding (homologous) part of the second parent's chromosome. However, in certain very rare and unpredictable instances, this recombination does not occur in the usual way. A gene duplication is an illegitimate recombination event that results in the duplication of a lengthy subsequence of a single chromosome. Susumu Ohno's seminal 1970 book *Evolution by Gene Duplication* proposed the provocative thesis that the creation of new proteins (and hence new organized structures and behaviors) begins with a gene duplication and that gene duplication is "the major force of evolution."

This paper describes six new architecture-altering genetic operations for genetic programming that are suggested by the mechanism of gene duplication (and the complementary mechanism of gene deletion) in chromosomes.

This paper proposes that these new operations be added to the toolkit of genetic programming when the user desires to evolve the architecture of a multi-part program containing automatically defined functions.

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 automatically and dynamically change the representation of the problem while simultaneously and automatically solving the problem.

Fourth, the new architecture-altering operations 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 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.

Section 2 of this paper describes the naturally occurring processes of gene duplication and gene deletion. Section 3 provides basic background information on genetic programming, automatically defined functions, and the five existing methods for determining the architecture of multi-part programs when automatically defined functions are being used. Section 4 describes the six new architecture-altering operations. Section 5 demonstrates that problems can be solved while the architecture is being evolved by showing examples of actual runs of genetic programming with the new operations.

2 GENE DUPLICATION AND DELETION IN NATURE

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. When 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 by the living cell as a result of a gene duplication.

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 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. But, 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. The second gene may accumulate additional changes and diverge more and more from the original gene. Eventually the modified gene may lead to the manufacture of a distinctly new and different protein that actually does affect the organism's structure and behavior in some advantageous or disadvantageous way. When a changed gene leads to the manufacture of a viable and advantageous new protein, natural selection again starts to work 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 claims that simple 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 distinctly 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.

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.

3 BACKGROUND ON GENETIC PROGRAMMING

Genetic programming is capable of evolving computer programs that solve, or approximately solve, various problems as demonstrated in the book *Genetic Programming: On the Programming of Computers by Means of Natural Selection* (Koza 1992). See also Koza and Rice 1992.

Many problem environments have regularities, symmetries, homogeneities, similarities, patterns, and modularities that can be exploited in solving the problem. An *automatically defined function* is a function (i.e., subroutine, `DEFUN`, procedure, module) that is dynamically evolved during a run of genetic programming and which may be called by a calling program (e.g., a main program) that is simultaneously being evolved. Automatically defined functions can be implemented within the context of genetic programming as described in the book *Genetic Programming II: Automatic Discovery of Reusable Programs* (Koza 1994a, 1994b).

Figure 1 shows an overall program consisting of one two-argument automatically defined function and one result-producing branch.

Before applying genetic programming to a problem, it is first necessary to perform at least five major preparatory steps. These steps involve determining (1) the set of terminals for each branch, (2) the set of functions for each branch, (3) the fitness measure, (4) the parameters for controlling the run, and (5) the result designation method and termination criterion.

In addition, when automatically defined functions are used, it is also necessary to perform a sixth major preparatory step concerning the architecture of the yet-to-be-evolved overall programs. This sixth step involves determining:

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

(b) the number of arguments 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.

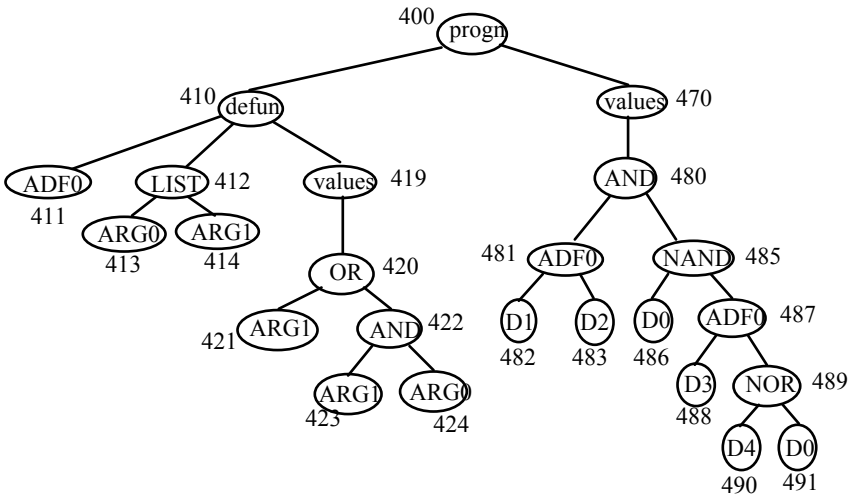

Figure 1: An overall computer program with one two-argument automatically defined function and one result-producing branch. The left branch beneath the `PROGN` contains the name of the automatically defined function (`ADF0`), its argument list (`ARG0` and `ARG1`), and its work-performing body (everything below the `VALUES` of `ADF0`). The right branch beneath the `PROGN` contains the work-performing body of the result-producing branch (everything below the second `VALUES`).

Sometimes these architectural choices flow so directly from the nature of the problem that they are obvious and virtually mandated. But, 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 minimum) number of arguments for each automatically defined function, or the optimal (or sufficient) arrangement of hierarchical references among the automatically defined functions.

The five existing methods (Koza 1994) for making these architectural choices include methods based on (1) prospective analysis of the nature of the problem, (2) seemingly sufficient capacity (overspecification), (3) affordable capacity, (4) retrospective analysis of the results of actual runs, and (5) evolutionary selection of the architecture.

The technique of evolutionary selection starts with an architecturally diverse initial random population. As the evolutionary process proceeds, certain individuals with certain architectures in the population will prove to be more fit than others in solving the problem. The more fit architectures prosper, while the less fit architectures tend to wither away. Eventually a program with a particular architecture may emerge that solves the problem. In this technique, various different architectures are created at the initial random generation (generation 0); however, no new architectures are ever created during the run and no architectures are altered during the run. There is a competition among the existing architectures during the course of the run.

4 THE NEW ARCHITECTURE-ALTERING 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, the architecture of the participating individuals change during the evolutionary process. Meanwhile, the Darwinian selection and the reproduction operation causes differential selection in favor of more fit individuals and individuals in the population are modified by the usual genetic operations of crossover and mutation.

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 new branch.

The step of selecting a program is performed 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.

The operation of branch duplication can now be illustrated by assuming that the program in Figure 1 has been selected as the program to participate in this operation. In step (3), a new function-defining branch (defining `ADF1`) is added to the selected program.

Figure 2 shows the program resulting after applying the operation of branch duplication to the program in Figure 1. Specifically, the function-defining branch 410 of Figure 1 defining `ADF0` (also shown as 510 of Figure 2) is duplicated.

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 of Figure 1 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 with the `ADF1` 581 in Figure 2. The arguments for the invocation of `ADF1` 581 are D1 582 and D2 583 in Figure 2 (i.e., they are identical to the arguments D1 482 and D2 483 for the invocation of `ADF0` at 481 as part 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.

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 1 is {2} because there is one two-argument function-defining branch. The program in Figure 2 has an argument map of {2, 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, the effect of this operation is to leave unchanged the value returned by the overall program.

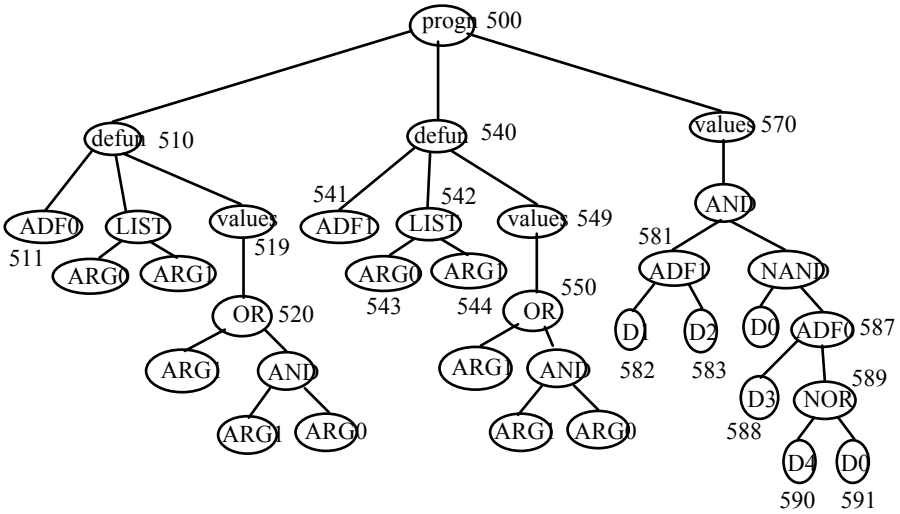

Figure 2: Program with two two-argument function-defining branches (defining `ADF0` and `ADF1`) of and one result-producing branch. `ADF1` results from application of the branch duplication operation to `ADF0` of the program in Figure 1.

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. `ADF0` and `ADF1` can be viewed as separate procedures for handling the two separate newly-created subproblems (cases). Immediately after the branch duplication operation, the two subproblems (cases) are handled in precisely the same way.

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 operations described herein) always produce a syntactically valid program.

Analogs of the naturally occurring operation of gene duplication have been used in connection with the genetic algorithm operating on character strings and other evolutionary algorithms for some time. Holland (1975, page 116) suggested that intrachromosomal gene duplication might provide a mean of adaptively modifying the effective mutation rate by making two or more copies of a substring of adjacent alleles. Lindgren (1991) analyzed the prisoner's dilemma game using an evolutionary algorithm that employed an operation analogous to gene duplication applied to chromosome strings.

Argument Duplication

The operation of *argument duplication* duplicates one of the arguments 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 in that invocation 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.

Figures 1 and 3 together illustrate the operation of argument duplication.

Suppose that `ARG1` labeled 414 in Figure 1 (also shown as 614 in Figure 3) is chosen as the argument-to-be-duplicated from this picked branch.

In Figure 3, the argument list of `ADF0` has been changed by adding a uniquely-named new argument, `ARG2` at 615. There are two occurrences of the argument-to-be-duplicated in the body of the picked function-defining branch of the selected program, namely at 423 and 424 in Figure 1. For each of these two occurrences, a random choice is made to either leave the occurrence of `ARG1` unchanged or to replace it with the newly created argument, `ARG2`. Figure 3 shows that the choice was in favor of a

replacement for the first occurrence of ARG1. Consequently, ARG2 now appears at 621 of Figure 3. As it happens, ARG1 remains unchanged at 623 of Figure 3.

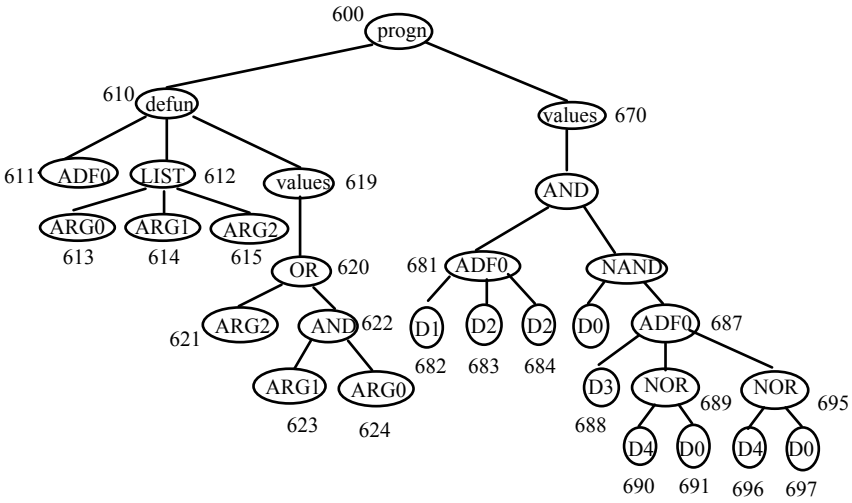

Figure 3: Program with one three-argument function-defining branches and one result-producing branch. ADF0 here results from application of the argument duplication operation to ADF0 of the program in Figure 1.

There are two occurrences of an invocation of ADF0 in the result-producing branch at 481 and 487 of Figure 1. In the first invocation of ADF0 at 481, the variable D2 483 corresponds to the argument-to-be-duplicated because it is the second argument of ADF0 481 and because it is the second argument (ARG1 414) that is the argument-to-be-duplicated. In the second invocation of ADF0 at 487, the entire argument subtree consisting of (NOR D4 D0) at 489, 490, and 491 corresponds to the argument-to-be-duplicated.

ADF0 681 and 687 now each take three arguments, instead of only two. For the first invocation of ADF0 at 681, D2 683 has been duplicated so that D2 now appears at both 683 and 684. For the second invocation of ADF0 at 687, the entire argument subtree (NOR D4 D0) has been duplicated so that it appears at both 689, 690, and 691 as well as 695. 696. and 697. The original program in Figure 1 has an argument map of {2} and the resulting program in Figure 3 has an argument map of {3}.

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.

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 of the selected program 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*) involves inserting the entire body of the branch-to-be-deleted for each instance of an invocation of that branch.

The first alternative (branch deletion by consolidation) begins by finding a suitable choice for the surviving branch within the overall program. When branch deletion by consolidation is performed, the number of arguments possessed by the proposed surviving branch may equal to, less than, or greater than the number of arguments possessed by the branch-to-be-deleted.

The simplest of these three possibilities for branch deletion by consolidation is the possibility where the number of arguments possessed by the proposed surviving branch is equal to the number of arguments possessed by the branch-to-be-deleted. This situation prevails in Figure 2. Suppose that the first function-defining branch (defining `ADF0`) of the program in Figure 2 is picked as the branch-to-be-deleted and that the second function-defining branch (defining `ADF1`) is to be the surviving branch. In that event, the first function-defining branch is deleted; the invocation of `ADF0` at 587 in Figure 2 is to be replaced by an invocation of `ADF1`; and the two argument subtrees below the invocation of `ADF0` at 587 are retained as the argument subtrees for the invocation at 587 of `ADF1`. That is, the branch-to-be-deleted is merged into `ADF1`. The original program in Figure 2 has an argument map of {2, 2} and the resulting program has an argument map of {2}.

For this simplest of the three possibilities, the branch deletion may be viewed as a generalization of a procedure. Before the branch deletion, the two function-defining branches constitute different subproblems (cases). The two branches do different things and they are invoked in different situations by the result-producing branch. After the branch deletion by consolidation, both subproblems (cases) are handled in the same way.

Consider the other two possibilities.

If the number of arguments required by the proposed surviving branch were less the number of arguments possessed by the branch-to-be-deleted, any superfluous argument subtrees below the invocation of the branch-to-be-deleted may simply be deleted. The branch deletion may also be viewed as a generalization of a procedure with an accompanying generalization of its arguments.

If the number of arguments required by a proposed surviving branch were greater than the number of arguments possessed by the branch-to-be-deleted, the required additional argument subtrees may be randomly generated. The random regeneration is accomplished using the same method of generation originally used to create the invoking branch (i.e., the branch containing the invocation of the branch-to-be-deleted) at the time of creation of the initial random population in generation 0 (with the branch-to-be-deleted being unavailable during this random regeneration). Random generation may not considered to be desirable because it introduces a significant mutational aspect to the operation. In that event, the operation may simply be aborted for this third possibility.

When the second alternative (branch deletion with random regeneration) is being used, all of the argument subtrees required by the invocation of the surviving branch are randomly generated.

The third alternative (branch deletion by macro expansion) has the advantage of preserving the semantics of the overall program; however, it has the disadvantage of usually creating very large programs. Each of the argument subtrees in each invocation of the branch-to-be-deleted is substituted into a copy of the body of the branch-to-be-deleted and the now-expanded body then replaces the invocation of the branch-to-be-deleted. If the objective of a deletion is change the semantics of the overall program (i.e., to achieve generalization), this alternative is undesirable.

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 anywhere in the body of picked function-defining branch of the selected program, replace the argument-to-be-deleted with a surviving argument (described below)

The operation of argument deletion may be viewed as a generalization in the sense that some information that was formerly considered in executing a procedure is now no longer considered.

When an argument is deleted, the question arises as to how to modify references to the argument-to-be-deleted within the picked branch.

One alternative (called *argument deletion by consolidation*) involves identifying another argument of the picked branch as the surviving argument and replacing (consolidating) the argument-to-be-deleted with the surviving argument in the picked branch.

A second alternative (called *argument deletion with random regeneration*) is to generate a new subtree in lieu of an invocation of the argument-to-be-deleted using the same method of generation originally used to create the picked branch at the time of creation of the initial random population (with the argument-to-be-deleted being unavailable during this random regeneration). The subtree may consist of either a single available argument or an entire generated argument subtree composed of the available functions and terminals.

A third alternative (called *argument deletion by macro expansion*) may also be used.

Suppose, in employing the first alternative (argument deletion by consolidation), that `ARG2` labeled 615 in Figure 3 is chosen as the argument-to-be-deleted and that `ARG1` is chosen (from among the remaining two arguments). The one occurrence of `ARG2` at 621 in Figure 3 is replaced by `ARG1`. The two invocations of `ADF0` by the result-producing branch (at 681 and 687) are modified by deleting the third argument subtree in each invocation. Specifically, the argument subtree D0 684 is deleted from the invocation of `ADF0` at 681 and the argument subtree (NOR D4 D0) at 695, 696, and 697 is deleted from the invocation of `ADF0` at 687. The result is the program shown in Figure 1. The original program in Figure 3 has an argument map of {3} and the resulting program in Figure 1 has an argument map of {2}.

Both the argument duplication and the branch duplication operations create larger programs. The argument deletion operation and the branch deletion operation can create smaller programs and provide a mechanism for balancing the continual growth that would otherwise occur (provided the alternative of argument deletion by macro expansion is not used).

Branch Creation

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

(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 an 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 complete.

(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.

This operation is, in a sense, a generalization of the operation of branch duplication.

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). First, Angeline and Pollack place each new function (called a module) created by the compression operation into a Genetic

Library so that the new function is not specifically associated with the selected program that gave rise to it. In contrast, in the branch creation operation, the new function may be invoked only by the selected program in which it was originally created and of which it is a part. A second difference is that the body of the new branch created by the branch creation operation continues to be subject to the effects of other operations in successive generations and is susceptible to continued change. A third difference is that the branch creation operation may be applied to any branch (and, in particular, to function-defining branches).

Argument Creation

The operation of argument creation creates a new argument within a function-defining branch of an overall program in a more general way than the previously described operation of argument duplication. This operation is, in some sense, a generalization of the operation of argument duplication.

The steps in the operation of argument creation are as follows:

(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 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 (e.g., the result-producing branch or other branch that invokes the picked function-defining branch), 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) in 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.

Creation of the Initial Random Population

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 minimal number of arguments appropriate to the problem. A second possibility is that each program in the population has a uniform architecture with no automatically defined functions (i.e., only a result-producing branch). The operation of branch creation is used 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 1994).

Structure-Preserving Crossover

When the new architecture-altering genetic operations are used, the population quickly becomes architecturally diverse (even if it was not initially created with architectural diversity). Structure-preserving crossover with point typing (as described in Koza 1994) permits robust recombination while simultaneously guaranteeing that any pair of architecturally different parents will produce syntactically and semantically valid offspring.

5 EXAMPLES OF ACTUAL RUNS

The architecture-altering operations described herein will now be illustrated by showing two actual runs of the problem of symbolic regression of the even-3-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). 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.

Example 1

The run starts with the random creation of a population of 1,000 individual programs using "single minimal ADF" approach (i.e., each program in generation 0 consists of one result-producing branch and a single one-argument function-defining branch and has an argument map of {1}).

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)))
```

This program was correct for six of the eight possible combinations (fitness cases) and thus the program scores a raw fitness of 6 (out of a possible 8).

The raw fitness of the best-of-generation program for generation 5 improves to 7. 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. The behavior of ADF1 is equivalent to the even-2-parity function.

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

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. The behavior of ADF2 consists of returning 1 only when ARG0 and ARG2 are 0 and ARG1 is 1.

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

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 can be verified as being 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 4 shows all of the ancestors of the just-described 100%-correct solution from generation 10 of this run 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 4. This figure 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 4 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 does not change the architecture (or argument map) of the offspring (relative to the mother).

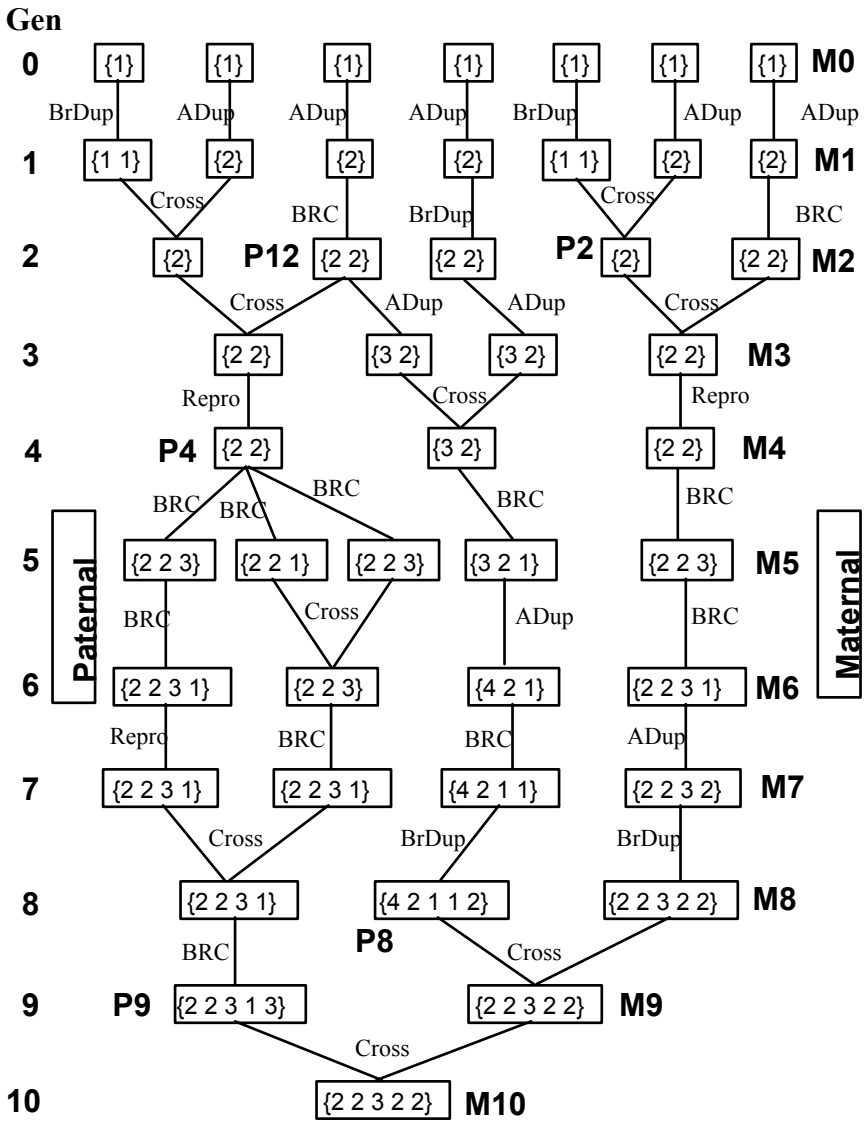

Figure 4: Complete genealogical audit trail showing all of the ancestors in generations 0 through 9 for the ultimate solution M10 from generation 10 for the run in example 1. The maternal line is shown at the right and the paternal line at the left.

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 4) 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 farther 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 4 has one fewer branches 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 4 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 4 has one fewer function-defining branch as program M6. Program M5 has an argument map of {2, 2, 3}. In turn, program M5 was the result of an branch creation from a single ancestor M4 from generation 4.

Program M4 from generation 4 shown on the right side of Figure 4 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 4 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 4) 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 4 has an argument map of {1} and has a raw fitness of 6. It has an argument map of {1} because the "single minimal ADF" approach is being used.

As can be seen, the problem of symbolic regression of the even-3-parity problem has been solved using the new architecture-altering operations within a run of genetic programming.

Example 2

A second run will illustrate the evolution of a hierarchical reference by one function-defining branch of another and illustrate the operation of branch deletion. In this run, a 100%-correct solution emerges in generation 15 to the problem of symbolic regression of the even-3-parity problem.

Figure 5 shows all of the maternal ancestors of the 100%-correct solution from generation 15 of this second run. Because 15 generations

are involved, Figure 5 (unlike Figure 4) does not show all of the ancestors. Figure 5 also shows the raw fitness of each program to the immediate left of the argument map of that program.

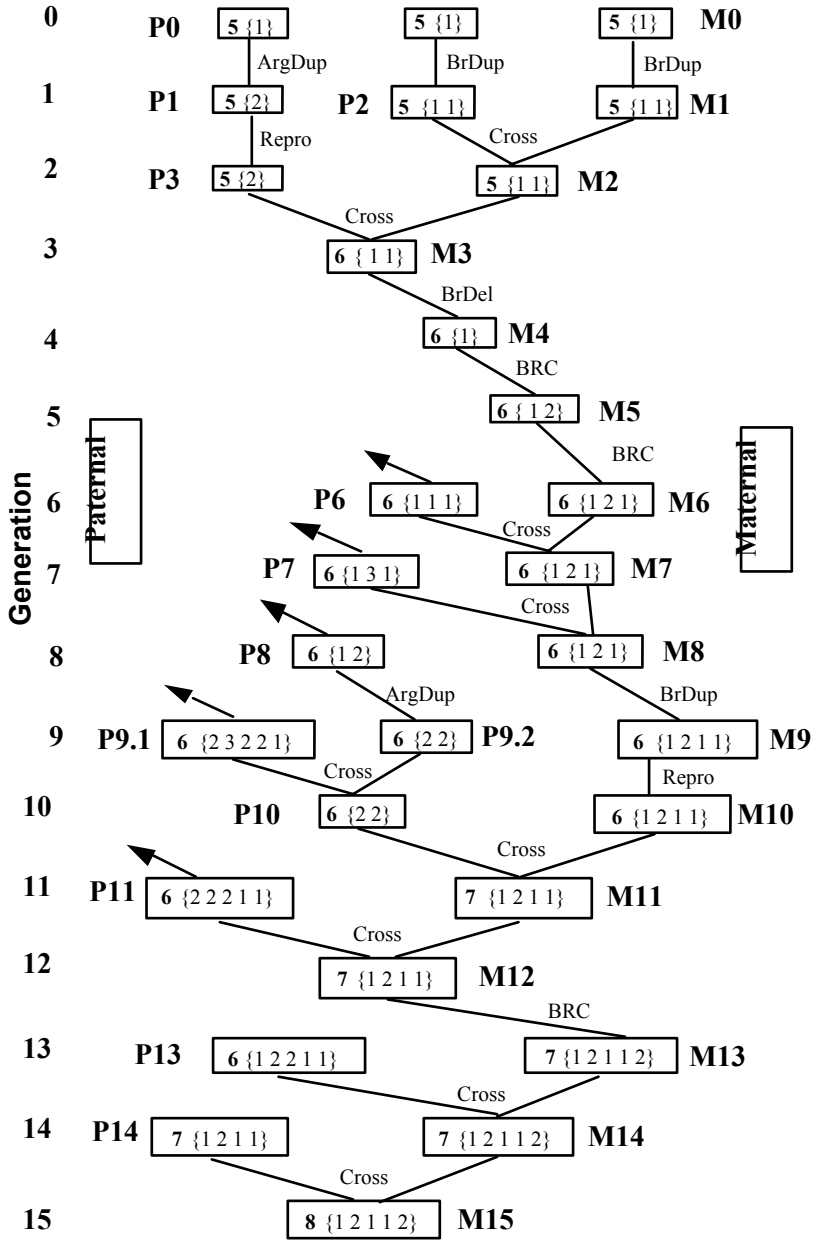

Figure 5: Partial genealogical audit trail showing all the maternal ancestors from generations 0 through 14 (and selected other ancestors) for the ultimate solution M15 from generation 15 for the run of example 2.

The best-of-generation program 3200 from generation 0 of this run has a raw fitness of only 5. There are many programs in the population with this level of fitness. This program 3200 is an early

ancestor of the 100%-correct solution that eventually emerges in generation 15. The result-producing branch of this program is shown below:

```
(OR (NOR D2 (ADF0 (AND D1 D0))) (AND D2 (ADF0 (NAND D0
           D0))))
```

ADF0 of this program is shown below:

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

ADF0 of this best-of-generation program from generation 0 is the one-argument negation (NOT) function. The one-argument NOT function was not one of the four primitive functions of the original problem (i.e., the two-argument AND, OR, NAND, and NOR functions).

A branch duplication operates on program 3200 to produce program 3201 in generation 1 with an argument map of {1, 1}.

Two crossovers occurring in generations 2 and 3 raise the raw fitness of the maternal ancestor 3203 in generation 3 from 5 to 6.

Program 3203 has two one-argument function-defining branches. Its ADF0 is "Always False" and is shown below:

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

ADF1 of this program 3203 is the exactly the same as ADF0 of ancestor 3200 at generation 0 (i.e., the NOT function). The branch duplication that created program 3201 in generation 1 duplicated ADF0 of ancestor 3200 of generation 0. Then the intervening crossovers modified ADF0 so as to convert it into the useless "Always False" function.

Then, a branch deletion removes the now-always-false ADF0 of program 3203 to produce program 3204 in generation 4. The surviving function-defining branch of program 3204 is exactly the same as ADF0 of ancestor 3200 at generation 0 (i.e., its it the NOT function). Program 3204 retains a fitness level of 6.

The result-producing branch of program 3204 from generation 4 is shown below:

```
(OR (NOR D2 (ADF0 (AND D1 D0))) (AND D2 (OR D1 D0)))
```

Next, a branch creation operation takes creates a new branch, ADF1, of program 3205 of generation 5 from the underlined portion of the result-producing branch of program 3204 above. The new branch, ADF1, is shown below:

```
(ADF0 (AND ARG1 ARG0))
```

The result-producing branch of program 3204 of generation 4 is also modified and the modified version is part of program 3205 of generation 5 as shown below:

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

Two crossovers, one reproduction, one branch creation, and one branch duplication then occur on the maternal lineage.

The ADF0 of program 3210 is the NOT function. ADF1 of program 3210 from generation 10 uses the hierarchical reference created above to emulate the behavior of the NAND function, as shown below:

```
(ADF0(AND ARG1 ARG0))
```

Mother 3210 mates with father 3230 to produce offspring 3211 in generation 11.

ADF0 of father 3230 performs the even-2-parity function and is shown below:

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

In the crossover, this entire branch from father 3230 was inserted into ADF1 of mother 3210 replacing (AND ARG1 ARG0) and producing a new ADF1 that performs the odd-2-parity function (since ADF0 performs the NOT function).

Three crossovers and one branch creation then occur on the maternal lineage; however, the ADF1 that was created in generation 11 and that performs the odd-2-parity functions remains intact.

The 100%-correct solution that emerged in generation 15 had an argument map of {1, 2, 1, 1, 2}. Only ADF0 and ADF1 of this particular program are referenced by the result-producing branch.

ADF0 performs the NOT function and is shown below:

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

ADF1 defines the odd-2-parity function by hierarchically referring to ADF0. ADF1 is shown below:

```
(ADF0 (AND (NAND (OR ARG1 ARG0) (AND (NAND ARG1 ARG0)
            (OR ARG1 ARG1))) (AND (NAND ARG0 (NOR ARG1
            ARG0)) (OR (AND ARG1 ARG1) (NOR ARG0
            ARG0)))))
```

Thus, a hierarchy of function definitions has emerged.

6 CONCLUSIONS

It is possible to evolve the architecture of a multi-part program using the new architecture-altering operations described herein while concurrently solving a problem in genetic programming.

More computational effort is required to perform the concurrent tasks of evolving the architecture while solving the problem than is required merely to solve the problem when the architecture is given. Future work will attempt to quantify the difference in these amounts of computation effort.

Acknowledgments

David Andre and Walter Alden Tackett wrote the computer program in C to implement genetic programming and five of the six new operations.

References

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

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.

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

Koza, John R. 1994. *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., and Rice, James P. 1992. *Genetic Programming: The Movie*. Cambridge, MA: The MIT Press.

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

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