TABLE OF CONTENTS

1. Introduction

In nature, crossover 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 chromosome. Susumu Ohno's seminal 1970 book *Evolution by Gene Duplication* proposed the provocative thesis that the creation of new proteins (and hence new structures and behaviors in living things) begins with a gene duplication and that gene duplication is "the major force of evolution."

This report 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 chromosome strings. This report 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 (ADFs) during the run of genetic programming.

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 report describes the naturally occurring processes of gene duplication and gene deletion. Section 3 describes analogs of gene duplication and gene deletion that have appeared in previous work with character strings in the field of genetic algorithms and other evolutionary algorithms. Section 4.1 provides basic background information on genetic programming and automatically defined functions. Section 4.2 lists the steps for executing genetic programming. Section 4.3 describes the five existing methods for determining the architecture of multi-part programs in the context of genetic programming with automatically defined functions. Section 4.4 describes different methods of creating the initial random population when these new operations are being used. Section 4.5 describes structure-preserving crossover with point typing in an architecturally diverse population. Section 5 describes the six new architecture-altering operations. Section 6 illustrates the architecture-altering operations using a *gedanken* experiment involving the problem of rotating the tires on an automobile. Section 7 contains some examples of actual runs of genetic programming with the new architecture-altering operations. Section 8 is the conclusion and section 9 outlines future work.

2. Gene Duplication and Deletion in Nature

In nature, deoxyribonucleic acid (DNA) is a long thread-like biological molecule that has the ability to carry hereditary information. DNA also has the ability to serve as a model for the production of replicas of itself. All known life forms on this planet (including bacteria, fungi, plants, animals, and humans) are based on the DNA molecule. The DNA molecule (called the chromosome) consists of a backbone that encases a long sequence of four nucleiotide bases: adenine (**A**), guanine (**G**), cytosine (**C**), and thymine (**T**).

Proteins are responsible for such a wide variety of biological structures and functions that it can be said that the structure and functions of living organisms are primarily determined by proteins (Stryer 1988). For example, some proteins transport particles such as electrons, atoms, or large macromolecules within living organisms (e.g., hemoglobin transports oxygen in blood). Some proteins store particular particles for later use (e.g., myoglobin stores oxygen in muscles). Some proteins generate nerve impulses (e.g., rhodopsin is the photoreceptor protein in retinal rod cells) while other proteins enable signals to be communicated in the nervous system. Some proteins provide physical structure (e.g., collagen gives skin and bone their high tensile strength). Other proteins create physical contractile motion (e.g., actin and myosin). Proteins are the basis of the immune system (e.g., antibodies recognize and combine in highly specific ways with foreign entities such as bacteria). Hormonal proteins transmit chemical instructions throughout the living organism. Other proteins control the expression of the genetic information contained in the nucleic acids that are responsible for the reproduction of the organism. Growth-factor proteins control growth and differentiation.

Protein molecules are polypeptides that are composed of sequences of between about 50 and several thousand amino acids. The sequence of amino acids appearing in a protein are specified by the sequence of nucleiotide bases appearing in the DNA. Sub-sequences consisting of three nucleiotide bases of DNA (called a codon) are translated, using the genetic code, into one of 20 amino acids.

Then, organisms consisting of proteins created in this manner spend their lives attempting to grapple with their environment. Some organisms in a given population do better than others in this pursuit. In particular, some organisms survive to the age of reproduction, reproduce a certain number of offspring, and thereby pass on all or part of their genetic make-up (their DNA) 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 that contribute to survival of the organism in its environment, that facilitate the survival of individual organisms to the age of reproduction, and that facilitate larger numbers of offspring.

As Charles Darwin stated in *On the Origin of Species by Means of Natural Selection* (1859),

> "I think it would be a most extraordinary fact if no variation ever had occurred useful to each being's own welfare ... . But if variations useful to any organic being do occur, assuredly individuals thus characterised will have the best chance of being preserved in the struggle for life; and from the strong principle of inheritance they will tend to produce offspring similarly characterised. This principle of preservation, I have called, for the sake of brevity, Natural Selection."

Biological populations display the ability to adapt, survive, and reproduce in their natural environments, to rapidly and robustly adapt in response to changes in the environment. Nature's methods for adapting biological populations to their environment and nature's method of adapting these populations to successive changes in their environments provides a potentially useful model for creating automated problem-solving techniques to problems that are generally thought to require “intelligence” to solve.

In nature, the naturally occurring genetic operations of mutation and crossover (sexual recombination) provide one way to alter the linear string of nucleiotide bases.

Mutation, for example, alters the chromosomal string by changing one nucleiotide base of the string. When the changed DNA is then translated into work-performing proteins in the living cell, the mutation may lead either to the manufacture of a variant of the original protein or, as is often the case, to no viable protein being manufactured from the altered portion of the DNA. The variant of the protein (or absence of the protein) may then affect the structure and behavior of the living thing in some advantageous or disadvantageous way. If the change is advantageous, natural selection will tend to perpetuate the change. In the more common situation where the random mutant is disadvantageous, natural selection will tend to cause the extinction of the mutant.

When crossover is performed, the linear string of nucleiotide bases of an offspring is created by recombining portions of the DNA of one parent with portions of the DNA of the second parent. The offspring produced by crossover usually differ from each parent by a substantial number of nucleiotide bases whereas the offspring produced by mutation differ from the single parent by only one base.

In addition to frequent changes introduced by mutation and crossover, chromosomes are occasionally also modified by other naturally occurring genetic operations, such as gene duplication and gene deletion.

Gene duplications are rare and unpredictable events in the evolution of genomic sequences. In gene duplication, there is a duplication of a portion of the linear string of nucleiotide bases of the DNA in the living cell. When a gene duplication occurs, there is no immediate change in the proteins that are manufactured by the living cell. The effect of a gene duplication is merely to create two identical ways of manufacturing the same protein. In the terminology of computer science, gene duplication is a semantics-preserving operation.

Then, over a period of time, 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 changing gene may be of no practical effect or value. In fact, the changed part of the linear string of nucleiotide bases of the DNA will often not even produce a viable protein. However, 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 continues as before. The changed gene is simply carried along in the DNA from generation to generation.

Natural selection exerts considerable force in favor of maintaining a gene that manufactures a protein that is important for the successful performance and survival of a living 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 the same 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 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 points out 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 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).

The moth *Chironomus tentans* provides an additional example of gene duplication in nature (Galli and Wislander 1993). In particular, we focus our attention on the particular contiguous sequence containing 3,959 nucleiotide bases of the DNA of this moth 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 moth'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.

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

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 shows the 253 amino acid residues of the C. tentans Sp38–40.A protein.

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

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 five residues of each protein are identical. That is, positions 240–244 of protein "A" are identical to positions 249–253 of protein "B." Since the proteins are of different length, identification of this particular similarity between the two proteins 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 similar 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 81% 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 moth.

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 IENEE-CDWT CDYEALPPPP GAKKDDKKEK    149
Second.protein    EKLECEKNAT P-GYKALFEF KESESFCEWE CDYEAI---P GAKKDEKKEK    146

First.protein     KIVKVVKPPK 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
```

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. Gene Duplication and Deletion in Evolutionary Algorithms

Analogs of the naturally occurring operation of gene duplication have been used in connection with the genetic algorithm and other evolutionary algorithms for some time.

Cavicchio (1970) used intrachromosomal gene duplication in early work on pattern recognition using the genetic algorithm (Goldberg 1989).

Holland (1975) 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. If there are k copies of an allele, the probability of a mutation of a particular allele is k times greater than if there were but one occurrence of the allele.

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 naturally occurring gene duplication.

The prisoner's dilemma is a problem in game theory with numerous psychological, sociological, and geopolitical interpretations. In this game, two players can either cooperate or not cooperate. The players make their moves simultaneously and without communication. Each player then receives a payoff that depends on his move and the simultaneous move of the other player. The payoffs in the prisoner's dilemma game are arranged so that a non-cooperative choice by one player always yields a greater payoff to that player than a cooperative choice (regardless of what the other player does). However, if both players are selfishly non-cooperative, they are both worse off than if they had both cooperated. The game is not a zero-sum game because, among other things, both players are better off if they both cooperate.

For a single encounter, the best strategy for each player is to be non-cooperative even though both are worse off than if they had both cooperated. However, the situation becomes considerably more interesting and complex if the two players engage in this game over a series of plays. In this so-called iterated version of the prisoner's dilemma, it becomes advantageous for cooperation to evolve (Axelrod 1984, 1987).

In Lindgren's work, strategies for playing the game over a series of plays are expressed as fixed-length binary character strings of length 2, 4, 8, 16, or 32. Strings of length 2 represent game-playing strategies that take account of only the one previous action by the opponent. For example, the string 01 instructs the player to make a non-cooperative move (indicated by a 0) if the opponent made an uncooperative move on his previous move and to make a cooperative move (indicated by 1) if the opponent just made a cooperative move. This particular strategy is called "tit-for-tat" since the player mimics his opponent's previous move. The string 10 is called "anti-tit-for-tat" because it instructs the player to do the opposite of what the opponent did on the previous move. String 11 is "Mr. Nice Guy" and 00 is "Darth Vader."

The 16 strategies represented by strings of length 4 take account of the player's own previous action as well as the opponent's previous action. Strings of length 8 look back even farther and take account of the opponent's action two moves ago in addition to both players' actions one move ago. Similarly, strings of length 16 and 32 take account of additional previous moves of the opponent and/or the player.

Lindgren used an evolutionary algorithm to evolve a population of game-playing strategies with varying degrees of look-back. Lindgren started with a population of

1,000 consisting of 250 copies of each of the 4 possible strings of length 2. The fitness of a string is measured according to the average score achieved by the strategy when that strategy is played interactively against all other strategies in the population. At each generational step of the process, a string is copied (reproduced) in proportion to its fitness.

The strings in the population are occasionally modified by Lindgren's evolutionary algorithm using three operations.

First, a mutation operation randomly alters a single bit in a single string.

Second, a gene duplication operation doubles a given character string. For example, the gene duplication operation transforms the string 01 into 0101. This operation has no immediate effect on the play because this lengthened string takes the previous move of the player himself into account, but then causes the very same action to be taken.

Third, a gene deletion operation (that Lindgren calls "split mutation") cuts the length of a string in half by randomly deleting either the first or second half of the string. For example, when this operation is applied to the string 1100, the result is either the string 11 or 00 (with equal probability). In general, this operation does have an immediate effect on the play.

Lindgren's evolutionary algorithm did not contain the crossover (recombination) operation.

Over a period of many generations, Lindgren (1991) found that the dynamics of this population of game-playing strategies exhibited many interesting evolutionary phenomena. Strategies with varying degrees of look-back spontaneously emerged, prospered, and became extinct. The phenomena of mass extinction and punctuated equilibrium were seen in the population as a whole while the evolutionary process progressed.

4. Background on Genetic Programming

John Holland's pioneering 1975 *Adaptation in Natural and Artificial Systems* described how the evolutionary process in nature can be applied to artificial systems using the genetic algorithm operating on fixed length character strings (Holland 1975). Additional information on current work in genetic algorithms can be found in Goldberg (1989), Forrest (1993), Davis (1987, 1991), and Michalewicz (1992).

Genetic programming is an extension of the genetic algorithm in which the genetic population consists of computer programs (that is, compositions of primitive functions, terminals, and possibly automatically defined functions).

Genetic programming starts with a primordial ooze of randomly generated computer programs composed of available programmatic ingredients and then genetically breeds the population of programs using the Darwinian principle of survival of the fittest and an analog of the naturally occurring genetic operation of crossover (sexual recombination). Genetic programming is described in the book *Genetic Programming: On the Programming of Computers by Means of Natural Selection* (Koza 1992). A videotape description of genetic programming can be found in Koza and Rice 1992. Genetic programming is capable of evolving computer programs that solve, or approximately solve, various problems from various fields. Many examples of recent work in genetic programming can be found in Kinnear (1994).

4.1. Background on Automatically Defined Functions

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, procedure, module, DEFUN) that is dynamically evolved during a run of genetic programming in association with a particular individual program in the population and which may be invoked 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 by establishing a constrained syntactic structure for the overall programs in the population. ADFs are described briefly in Koza 1992 and more extensively in the book *Genetic Programming II: Automatic Discovery of Reusable Programs* (Koza 1994a). A videotape description of automatically defined functions can be found in Koza 1994b.

When automatically defined functions are being used, each program in the population contains one or more function-defining branches (each defining one 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 and one result-producing branch.

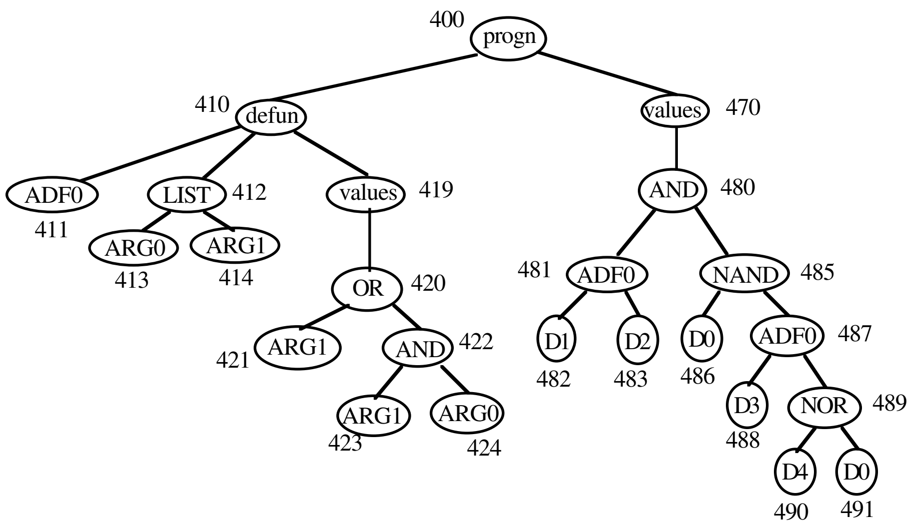

Figure 1: A program with one two-argument automatically defined function (ADF0) and one result-producing branch with an argument map of {2}.

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 function-defining branch that takes two arguments.

The program in figure 1 contains architecture-defining points (also sometimes called "invariant points" because they are not altered by crossover or mutation) 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 1 were to have more than one automatically defined functions, there would be additional occurrences of items (2), (3), (4), (5), and (6) for each additional function-defining branch.

The program in figure 1 also contains work-performing points (also sometimes called "noninvariant points" because these points are almost always different from branch to branch within a program and from program to program within the population). These work-performing points are the bodies of the result-producing branch and the function-defining branch(es).

The work-performing points of figure 1 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).

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 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. In figure 1, 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 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 1, 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.).

4.2. Steps for Executing Genetic Programming

Genetic programming is a domain-independent method that genetically breeds populations of computer programs to solve problems.

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 of computer programs.

(2) Iteratively perform the following sub-steps until the termination criterion 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.

4.3. Methods of Determining the Architecture of a Multi-Part Program

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 first necessary to perform a sixth major preparatory step. The sixth major step concerns the architecture of the yet-to-be-evolved overall programs in the population and involves determining

(a) the number of function-defining branches,

(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.

Sometimes these architectural choices flow so directly from the nature of the problem that they are obvious and virtually mandated. 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 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 1994a) 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.

4.3.1. Method of Prospective Analysis

Some problems can be analyzed and decomposed into subproblems of known dimensionality. This insight or information can then be used to establish a common architecture for all programs in the population.

For example, some problems involve finding a computer program (i.e., a mathematical expression, a composition of primitive functions and terminals) that produces the observed value of a dependent variable as its output when given the values of a certain number of independent variables as input. Problems of this type are called problems of symbolic regression, system identification, or simply "black box" problems. In many instances, it may be known that a certain number of the independent variables represent a certain relevant subspace or subsystem. In that event, the problem may be decomposed into subproblems based on the known (usually lower) dimensionality of the subspace or subsystem. When applying genetic programming with automatically defined functions to a problem where one discerns a subproblem of a certain dimensionality, one can use this insight to make the number of arguments of at least one of the function-defining branches equal to this dimensionality. Also, if it is known that there are a certain number of subspaces or subsystems, one could use this insight to choose that number as the number of function-defining branches. In practice, exact knowledge of these numbers is unnecessary; upper bounds on these numbers can be used to make the choice of the number of function-defining branches and the number of arguments possessed by each function-defining branch.

4.3.2. Method of Seemingly Sufficient Capacity

For many problems, the choice of the common architecture for the programs in the population can be made on the basis of providing seemingly sufficient capacity. That

is, one over-specifies the number of function-defining branches and the number of arguments possessed by each function-defining branch. Over-specification often works because genetic programming with automatically defined functions exhibits considerable ability to ignore extraneous dummy arguments of an automatically defined function, to ignore extraneous automatically defined functions, and to ignore unproductive hierarchical references among already-defined automatically defined functions.

4.3.3. Method of Using Affordable Capacity

Resources are required by each additional function-defining branch and each additional argument, especially if the function-defining branches are permitted to invoke one another hierarchically. Thus, the practical reality is the amount of resources that one can afford to devote to a particular problem will strongly influence or dictate the choice of the common architecture for the programs in the population. Often the architectural choice is necessarily made on the basis of hoping that the resources that one can afford to devote to the problem will prove to be sufficient to solve the problem.

4.3.4. Method of Retrospective Analysis

A retrospective analysis of the results of sets of actual runs made with various architectural choices can determine the optimal architectural choice for a given problem. The idea is to make a number of runs of the problem with different combinations of the number of function-defining branches and the number of arguments that they each possess, to retrospectively compute the effort required to solve the problem with each such architecture, and to identify the optimal architecture. If one is dealing with a series of related problems, a thorough retrospective analysis of one problem may provide guidance for making the required architectural choice for a similar problem.

4.3.5. Evolutionary Selection of the Architecture

The fifth technique for establishing the architecture of the overall program for solving a problem is to evolutionarily select the architecture dynamically during the run of genetic programming (described in Koza 1994a, chapters 21 – 25).

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.

The architecturally diverse populations used with the technique of evolutionary selection require a modification of both the method of creating the initial random population (described in the next section below) and the crossover operation (described in the succeeding section below). Modifications are also required when the six new architecture-altering operations are used.

4.4. Creation of the Initial Random Population in an Architecturally Diverse Population

The initial random population of programs may be created in any one of several possible ways when the six new architecture-altering operations described below are being used.

One possibility (called the "minimalist approach") is that each multi-part program in the initial random population at generation 0 has a uniform architecture with exactly one automatically defined function possessing the minimal number of arguments appropriate to the problem. For example, for a problem involving floating-point variables, the argument map of every individual in generation 0 would usually be {1} under the "minimalist approach".

A second possibility is that each program in the initial population has a uniform architecture with no automatically defined functions (i.e., only a result-producing branch). That is, the argument map of every individual in generation 0 is {}. In this event, the operation of branch creation must used to create function-defining branches. This operation may then be used with an unusually high frequency on generation 0 (where it is called the "big bang") to rapidly introduce multi-part programs into the population.

A third possibility is that the population at generation 0 is architecturally diverse. This is the approach used when the technique of evolutionary selection of the architecture is being used. In this approach, the creation of an individual program in the initial random population begins with a random choice of the number of automatically defined functions, if any, that will belong to the program. Then a series of independent random choices is made for the number of arguments possessed by *each* automatically defined function, if any, in the program. All of these random choices are made within a wide range that includes every number that might reasonably be thought to be useful for the problem at hand. Zero is included in the range of choices for the number of automatically defined functions, so the initial random population also includes some programs without any automatically defined functions. Once the number of automatically defined functions is chosen for a particular overall program, the automatically defined functions, if any, are systematically named in the usual sequential manner from left to right.

The range of possibly useful numbers of arguments for the automatically defined functions cannot, in general, be predicted with certainty for an arbitrary problem. There are some problems involving only a few actual variables where it is useful to have an automatically defined function that takes a large number of arguments. However, most problem-solving efforts focus primarily on solving problems by decomposing them into problems of lower dimensionality. Accordingly, it may be reasonable to cap the range of the number of probably-useful arguments for each automatically defined function by the number of actual variables of the problem. There is no guarantee that this cap (motivated by the desire to decompose problems) or any other cap is necessarily optimal, desirable, or sufficient to solve a given problem. In any event, practical considerations concerning resources often play a controlling role in setting the upper bound on the number of arguments to be permitted.

The range of potentially useful numbers of automatically defined functions cannot, in general, be predicted with certainty for an arbitrary problem. The number of potentially useful automatically defined functions does not necessarily bear any relation to the dimensionality of the problem. However, once again, considerations of resources play a controlling role in setting the upper bound on the number of automatically defined functions to be permitted. In practice, we often cap the number of automatically defined functions at the number of actual variables of the problem.

In practice, a zero-argument automatically defined function may or may not be a meaningful option. In the floating-point, integer, and certain other domains, it may be useful to include random constants because a zero-argument automatically defined function can be used to create an evolvable constant that can then be repeatedly called from elsewhere in the overall program. However, in the special case of the Boolean domain, the two possible Boolean constants (`T` or `NIL`) have limited usefulness because all compositions of these two constants merely evaluate to one of these two values. If an automatically defined function has no access to the actual variables of the problem, has no dummy variables, does not contain any side-effecting primitive functions, and does not contain any random constants, nothing is available to serve as terminals (leaves) of the program tree in the body of such a zero-argument automatically defined function.

Random choices occurring during the creation of the initial random population determine whether the body of any particular function-defining branch of any particular program in the population actually hierarchically calls all, none, or some of the automatically defined functions that it is theoretically permitted to call. Subsequent crossovers may, of course, change the body of a particular function-defining branch during the run and thereby change the automatically defined functions that a branch actually calls hierarchically. Thus, the function-defining branches have the ability to organize themselves into arbitrary disjoint hierarchies of dependencies among the available automatically defined functions. For example, within an overall program with five automatically defined functions at generation 0, `ADF4` might actually refer only to `ADF2` and `ADF3`, with `ADF2` and `ADF3` not referring at all to either `ADF0` or `ADF1`. Meanwhile, `ADF1` might refer only to `ADF0`. In this situation, there would be two disjoint hierarchies of dependencies. A subsequent crossover might change this organization. For example, after such a crossover, `ADF3` might refer to `ADF0`, but still not to `ADF1`, thereby establishing a different hierarchy of dependencies. Any allowable (i.e., noncircular) hierarchy of dependencies may thus be created in generation 0 or created by crossover during the evolutionary process.

4.5. Structure-Preserving Crossover in an Architecturally Diverse Population

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.

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.

In the simplest practical implementation of genetic programming (as exemplified throughout most of Koza 1992), no syntactic constraints are involved in constructing individual programs in the population (except for a syntactic constraint as to the overall size of the program). Program size is typically measured in terms of depth (or, alternately, in terms of the total number of points in the program). All points in all programs are of a single common type. Subject only to the size limit established for generation 0, any function from the function set of the problem and any terminal from the terminal set may appear at any point in a program during the initial random creation of programs in generation 0. Similarly, subject only to the size limit established for offspring, the crossover operation produces valid offspring regardless of what points are chosen as crossover points from the two parents.

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 (invariant) 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 (noninvariant) 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 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 (described below) 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. The Six New Architecture-Altering Genetic Operations

The six new architecture-altering genetic operations are as follows:

(1) *Branch duplication* creates one new offspring program for the new population by duplicating a function-defining branch of an existing program (and making additional appropriate changes in the program to reflect the duplication).

(2) *Argument duplication* creates 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 in the program to reflect the duplication).

(3) *Branch deletion* creates one new offspring program for the new population by deleting one function-defining branch of one existing program (and making additional appropriate changes in the program to reflect the deletion).

(4) *Argument deletion* creates 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 in the program to reflect the deletion).

(5) *Branch creation* creates 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.

(6) *Argument creation* creates 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.

The six new architecture-altering operations differ from the operations of reproduction crossover, and mutation in that the argument maps of the participating individuals change as a consequence of performing the operations.

During each generation of the evolutionary process, a certain percentage of the individuals in the population participate in the architecture-altering operations. Meanwhile, Darwinian selection causes differential selection in favor of more fit individuals. And, during each generation, individuals in the population are modified by the operations of crossover and mutation. The result is that the evolutionary process will select against individuals in the population with architectures that are less suitable for solving the problem. Individuals with unsuitable architectures will tend to become extinct over a period of generations. Similarly, individuals with architectures that facilitate solving the problem will tend to prosper.

The six new operations are described in detail below.

5.1. 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. If the selected program has only one function-defining branch, that branch is automatically picked. If the selected program has no function-defining branches (or already has the maximum number of branches established for the problem at hand), this operation is aborted and no action is performed on this occasion.

(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. If the choice is to make the replacement, the arguments in the invocation of the new branch remain identical to the arguments of the existing invocation.

The step of selecting a program is performed on the basis of fitness for branch duplication (and the other operations subsequently described herein), 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. A copy is first made of the selected program and the operation is then performed on the copy, so the unchanged original program remains in the population and is therefore available to be selected again on the basis of its fitness.

The problem of symbolic regression of the Boolean even-parity function will be used for purposes of illustration throughout this report. The Boolean even-k-parity function takes k Boolean arguments, `D0`, `D1`, `D2`, and so forth (up to a total of k arguments). Each argument can take on the value `T` (true or 1) or `NIL` (false or 0). The even-k-parity function returns `T` if an even number of its Boolean arguments are `T`, but otherwise returns `NIL`. Parity functions are used to check the accuracy of stored or transmitted binary data in computers because a change in the value of any one of its arguments always changes (toggles) the value of the function. Because of this toggling, parity functions are often used as benchmarks in the study of machine learning and neural networks. The problem is to discover a 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.

Suppose that the program in figure 1 has been selected, in step (1), as the program to participate in the operation of branch duplication. Since this program happens to have only one function-defining branch, the sole function-defining branch (defining `ADF0`) is picked, in step (2), as the branch-to-be-duplicated.

In step (3), a new function-defining branch is added to the selected program, thus increasing, by one, the number of function-defining branches in the selected program. The new branch is given the new name of `ADF1`. This new name is unique within this program.

Figure 2 shows the program resulting after applying the operation of branch duplication to the program in figure 1. The original program in figure 1 has an argument map of {2}. The program in figure 2 has an argument map of {2, 2} because the operation of branch duplication duplicated the two-argument function-defining branch.

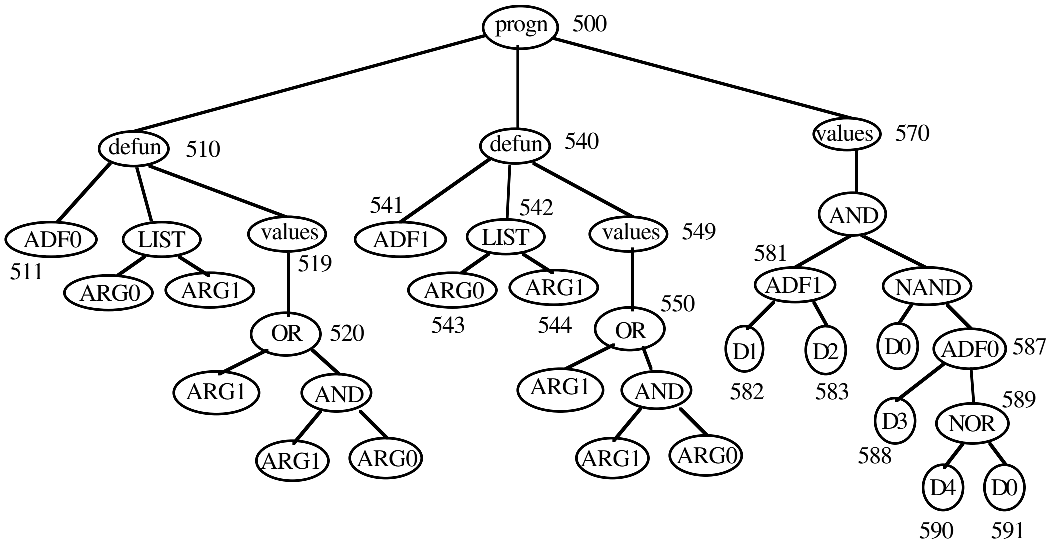

Figure 2: Program with argument map of {2, 2} created using the operation of branch duplication.

Specifically, the function-defining branch starting at the DEFUN labeled 410 of figure 1 defining ADF0 (also shown as 510 of figure 2) is duplicated. The duplicated branch is added at DEFUN 540 of figure 2, thereby giving the new overall program three branches: a first function-defining branch starting at DEFUN 510, a new second function-defining branch starting at DEFUN 540, and a result-producing branch starting at VALUES 570. The new function-defining branch is given the unique new name, ADF1, at 541. The argument list of the new function-defining branch is the same as the argument list of the branch-to-be-duplicated and consists of ARG0 543 and ARG1 544. The body of the new function-defining branch starting with VALUES 549 is the same as the body of the first function-defining branch starting at VALUES 419 of figure 1 (also shown as 519 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 481 and ADF0 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 made to replace ADF0 with ADF1. Thus, ADF1 appears at 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 in figure 1). For the second invocation of ADF0 at 487 of figure 1, the choice is randomly made to leave ADF0 unchanged. Thus, ADF0 appears at 587 of figure 2.

Because the duplicated new function-defining branch is identical to the previously existing function-defining branch (except for its name) and because the new function-defining branch ADF1 is invoked with the same arguments as ADF0 had been invoked, the value returned by the overall program is unchanged by the operation of branch duplication. The programs are semantically equivalent but, of course, structurally (syntactically) different.

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 newly-created procedures for handling

the two separate 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. These subsequent changes can 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 (and the other operations subsequently described herein) always produce a syntactically valid program.

The operation of branch duplication (and the operations of argument duplication, branch creation, and argument creation described below) are recombinative in the sense that the offspring produced by each operation consists entirely of genetic material that comes from an existing member of the population.

5.2. 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. If the selected program has only one function-defining branch, that branch is automatically chosen. If the selected program has no function-defining branches, this operation is aborted and no action is performed on this occasion.

(3) Choose one of the arguments of the picked function-defining branch of the selected program as the argument-to-be-duplicated. If the picked function-defining branch has no arguments (or already has the maximum number established for the problem at hand), this operation is aborted and no action is performed on this occasion.

(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 the 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 (e.g., the result-producing branch or other branch that invokes the picked function-defining branch), 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 value returned by the overall program is unchanged by the operation of argument duplication.

Figures 3 and 1 together illustrate the operation of argument duplication. The original program in figure 1 has an argument map of {2}. The new program in figure 3 has an argument map of {3} because the operation of argument duplication added an argument to the one function-defining branch.

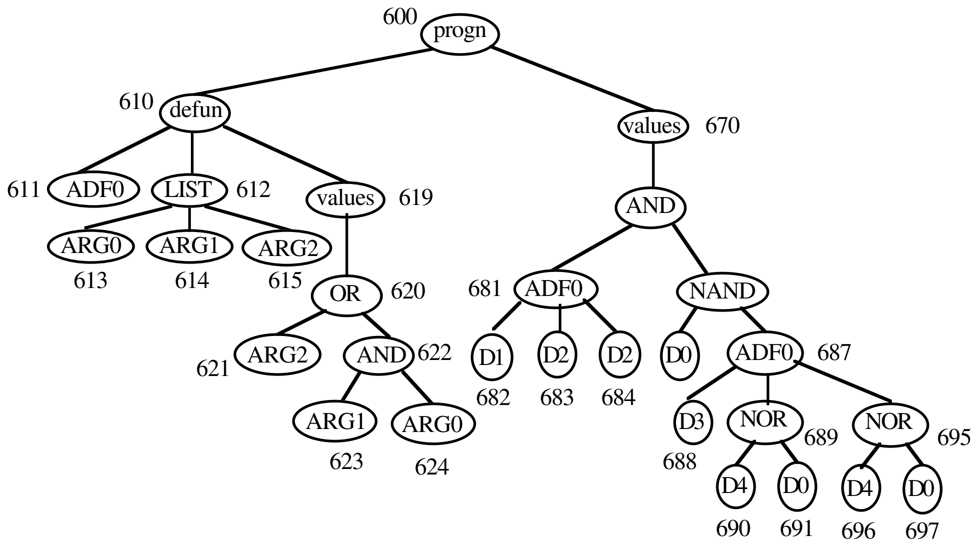

Figure 3: Program with argument map of {3) created using the operation of argument duplication.

ADF0 in figure 1 takes two dummy arguments, namely ARG0 (labeled 413) and ARG1 (labeled 414). Now suppose that the second argument, ARG1 414 in figure 1 is chosen as the argument-to-be-duplicated.

In figure 3, the argument list of ADF0 has been changed by adding a uniquely-named new argument, ARG2 at 615, thereby increasing the size of this argument list from two to three. 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 421 and 423 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 at 421. Consequently, the new name, ARG2, appears at 621 of figure 3. The choice was against replacement for the second occurrence of ARG1 at 423 of figure 1, so ARG1 appears at 623 of figure 3.

There are two occurrences of an invocation of ADF0 in the result-producing branch at 481 and 487 of figure 1. The second argument, ARG1, is the argument-to-be-duplicated in this example. In the first invocation of ADF0 at 481, the variable D2 (labeled 483) corresponds to the argument-to-be-duplicated because it is the second argument of ADF0 (labeled 481). 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.

Because of the argument duplication, ADF0 681 and ADF0 687 in figure 3 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 of figure 3. 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 of figure 3.

Just as the operation of branch duplication was interpreted as a "case splitting," the operation of argument duplication can be interpreted as a "case splitting." Immediately after the argument duplication operation, the two subproblems (cases) are handled in precisely the same way. The particular instantiations of the second and third arguments in each invocation of ADF0 provide the opportunity for the subsequent evolution of different ways of handling the two separate subproblems (cases).

Subsequent genetic operations may alter one or both of these two presently-identical arguments and these subsequent changes can lead to a divergence in structure and behavior. Once the second and third arguments diverge, this divergence may be interpreted as a specialization or refinement.

5.3. 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 of the selected program as the branch-to-be-deleted. If the selected program has no function-defining branches, this operation is aborted and no action is performed on this occasion.

(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 (e.g., the result-producing branch or other branch that invokes the branch-to-be-deleted), 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. We consider three alternatives.

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 almost never preserves the semantics of the overall program. 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. Branch deletion with random regeneration almost never preserves the semantics of the overall program.

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. Branch deletion by macro expansion preserves the semantics of the overall program at the expense of a massive increase in the size of the overall program.

5.3.1. Branch Deletion by Consolidation

The first alternative (branch deletion by consolidation) begins by finding a suitable choice for the surviving branch within the overall program. Since branch deletion by consolidation involves two branches (i.e., the branch-to-be-deleted and the surviving branch), it is necessary that the selected program have at least two function-defining branches. Thus, when branch deletion by consolidation is being used and the selected program has less than two function-defining branches, this operation is aborted and no action is performed on this occasion.

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.

Figure 2 illustrates the first of these three possibilities for branch deletion by consolidation (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). 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 first possibility, the branch deletion may be viewed as a generalization of a procedure. Before the branch deletion, the two function-defining branches constitute different procedures for handling 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. That is, the procedure is generalized so that the surviving branch (ADF1 here) handles both subproblems (cases).

In the second possibility, the number of arguments required by the proposed surviving branch is less then the number of arguments possessed by the branch-to-be-deleted. Any superfluous argument subtrees below the invocation of the branch-to-be-deleted are simply deleted. The branch deletion may also be viewed as a generalization of a procedure with an accompanying generalization of its arguments.

In the third possibility, the number of arguments required by the a proposed surviving branch is greater than the number of arguments possessed by the branch-to-be-deleted. The required additional argument subtrees may be randomly generated (or duplicated). In the later case, the random creation is done 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). This approach may not 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.

5.3.2. Branch Deletion with Random Regeneration

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

Except when random regeneration is used, the operation of branch deletion (and the operation of argument deletion described below) are recombinative in the sense that the offspring produced by each operation consists entirely of genetic material that comes from an existing member of the population. When random regeneration is used, branch deletion (and argument deletion) are partially recombinative and partially mutational.

5.3.3. Branch Deletion by Macro Expansion

The third alternative (branch deletion by macro expansion) has the characteristic of preserving the semantics of the overall program; however, it has the disadvantage of

usually creating very large programs. Moreover, branch deletion by macro expansion would not work if recursion were being used, 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. Of course, if the objective of a deletion is change the semantics of the overall program (i.e., to achieve generalization), this alternative is undesirable.

5.4. 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. If the selected program has only one function-defining branch, that branch is automatically picked. If the selected program has no function-defining branches, this operation is aborted and no action is performed on this occasion.

(3) Choose one of the arguments of the picked function-defining branch of the selected program as the argument-to-be-deleted. If the picked function-defining branch has no arguments (or already has only the minimum number of arguments established for the problem at hand), this operation is aborted and no action is performed on this occasion.

(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 (e.g., the result-producing branch or other branch that invokes the picked function-defining branch), 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 the picked function-defining branch of the selected program, replace the argument-to-be-deleted with a surviving argument (described below).

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

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. When this alternative is employed, 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.

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).

A third alternative (called *argument deletion by macro expansion*) may also be used. This alternative has the advantage of preserving the semantics of the overall program; however, it has the disadvantage of usually creating large programs.

5.4.1. Argument Deletion by Consolidation

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) as the surviving argument. 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}.

Since argument deletion by consolidation involves two arguments (i.e., the argument-to-be-deleted and the surviving argument), it is necessary that the picked branch have at least two arguments. Thus, when argument deletion by consolidation is being used and the picked function-defining branch has less than two arguments, this operation is aborted and no action is performed on this occasion.

It is often advisable to set a minimum permissible number of arguments for any function. For example, in a problem involving Boolean functions and no side-effects, there are only four possible Boolean functions of one argument, so it may be more efficient to exclude such one-argument functions for a Boolean problem. If this approach is adopted, whenever the picked function-defining branch has less than three arguments, the operation of argument deletion is aborted and no action is performed on this occasion.

5.4.2. Argument Deletion with Random Regeneration

When the second alternative (argument deletion with random regeneration) is being used, a new subtree is randomly generated 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.

5.4.3. Argument Deletion by Macro Expansion

When argument deletion by macro expansion (i.e., the third alternative) is used, the first step is to delete the argument-to-be-deleted from the argument list of the picked branch. The second step is then to create as many copies of the now-modified picked branch as there are invocations of the picked branch in the overall program (e.g., in the result-producing branch or other branches) and to give each such copy of the picked branch a unique name. The third step is to replace each invocation of the picked branch in the overall program with an invocation to a particular one of the uniquely-named copies of the now-modified picked branch. The fourth step is, for each uniquely-named copy of the now-modified picked branch, to insert the argument subtree corresponding to the argument-to-be-deleted for every occurrence of the argument-to-be-deleted in that particular copy.

The alternative of argument deletion by macro expansion has the characteristic of preserving the semantics of the overall program; however, it has the disadvantage of usually creating a vast number of additional branches and large overall programs. Of course, if the objective of a deletion is change the semantics of the overall program (i.e., to achieve generalization), this alternative is undesirable.

Both the argument duplication and the branch duplication operations create larger programs. Larger programs consume more resources and a population of such growing programs may become unmanageable as a practical matter. 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).

5.5. Branch Creation

The operation of branch creation creates a new automatically defined function within an overall program in a more general (but less biologically motivated) way than the previously described operation of branch duplication.

The steps in the operation of branch 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 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 (e.g., in a depth-first manner).

(4) As each point below the picked point in the picked branch 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 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 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 picked branch 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 defined by the new function-defining branch.

The operation of branch creation is more general than the operation of branch duplication in the sense that the picked point may be in the result-producing branch and also in the sense that the picked point need not be the top-most point of the body of an existing branch.

The operation of branch creation described herein is similar to, but different from, the compression (module acquisition) operation described by Angeline and Pollack in at least three ways (Angeline and Pollack 1993, 1994; Angeline 1994a, 1994b).

First, Angeline and Pollack place each new function (called a module) created by the compression operation into a Genetic Library. The new function is not exclusively associated with the selected program that gave rise to it. Instead, new functions placed in the Genetic Library may be invoked by any program in the population. In practice, this occurs as soon as crossover or some other operation introduces references to the new function. In contrast, in the branch creation operation described herein, each new function is made a part of the selected program as a new branch of that particular program. 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 between the compression operation described by Angeline and Pollack and operation of branch creation described herein is that the body of the new branch created by the branch creation operation continues to be subject to the effects of other operations (notably crossover) in successive generations of the process. Thus, instead of being insulated from future change by residing in the Genetic Library, it is susceptible to continued change. Among the possible changes that may occur by means of crossover are the insertion of additional hierarchical references to other automatically-defined functions.

A third difference between the compression operation described by Angeline and Pollack and operation of branch creation described herein is that the branch creation operation may be applied to any branch (and, in particular, to function-defining branches). Angeline and Pollack's operation was limited to one a single 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. In *depth compression* described by Angeline and Pollack, the points at a certain distance (depth) below the picked point are designated. In *leaf compression* described by Angeline and Pollack, all of the external points (leaves) below the picked point are designated. Other methods of designation may be used. Here internal points below the picked point are designated independently at random with a certain probability. Regardless of the method of designation, zero points, one point, or several points are designated.

Figures 4 and 1 illustrate the operation of branch creation.

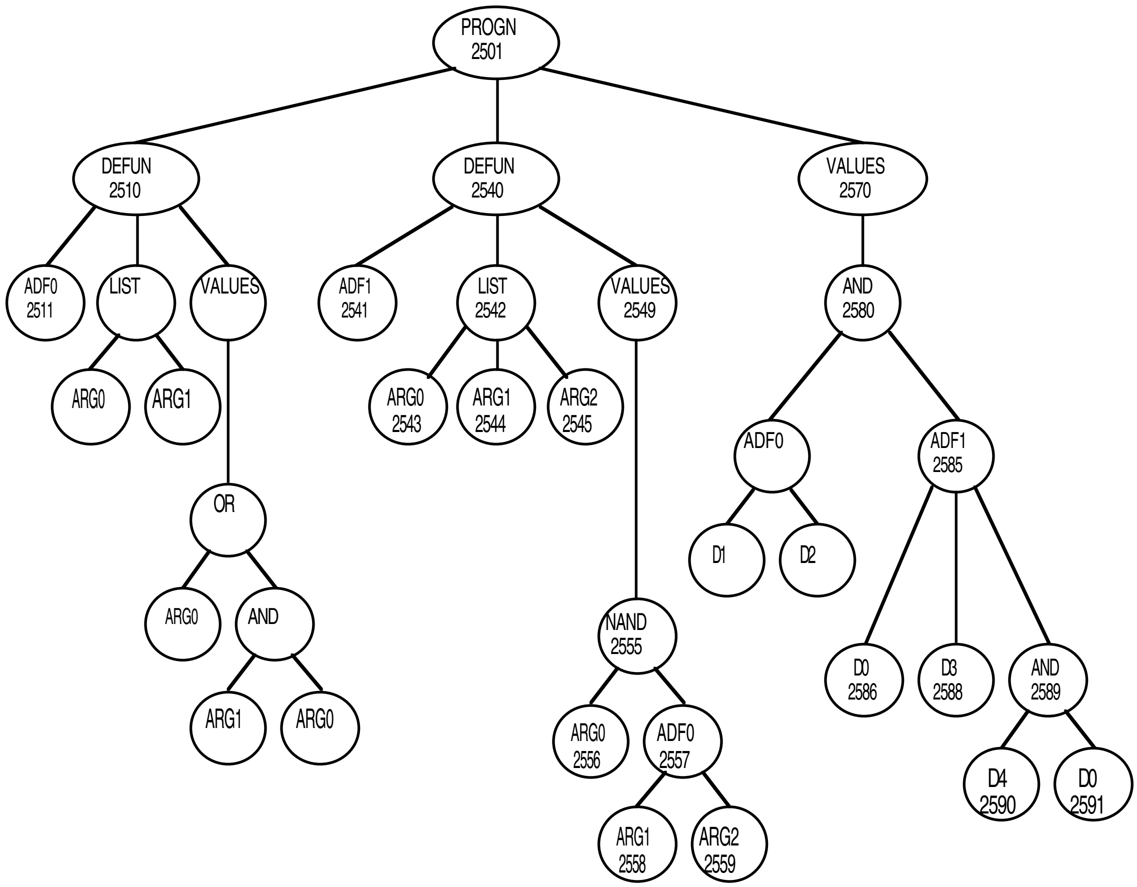

Figure 4: Program with argument map of {2, 3}.

Suppose the point NAND 485 is picked from the result-producing branch of the program starting at PROGN 400 in figure 1. Starting at the picked point, NAND 485, a depth-first traversal of the subtree below this point will visit 486, 487, 488, 489, 490 and 491 of figure 1 in that order. Suppose, while making this depth-first traversal, points 486, 488, and 489 of figure 1 are designated. The fact that three points were designated means that the branch-to-be-created will be a three-argument function-defining branch. The three argument subtrees starting at these three designated points will become arguments for the branch-to-be-created. Because point 489 was designated, no traversal of D4 490 and D0 491 is made.

The branch creation operation causes a new branch to be added to the overall selected program. The new branch starts at DEFUN (labeled 2540) in figure 4. The new branch is given a new name, ADF1 (labeled 2541), that is unique within the new overall program. The argument list of this new branch in figure 4 contains three consecutively-numbered dummy variables (formal parameters), namely ARG0 (labeled 2543), ARG1 (labeled 2544), and ARG2 (labeled 2545) appearing below LIST (labeled 2542).

The body of this new branch starts at VALUES (labeled 2549). The body of the new branch in figure 4 consists of a modified copy of the seven-point subtree starting at the picked point, NAND 485 of figure 1. In modifying the copy, each of the three designated points from figure 1 (486, 488, and 489) is replaced by a different consecutively-numbered dummy variable, ARG0, ARG1, and ARG2 in figure 4. Specifically, the first designated point, D0 486, from figure 1 is replaced by ARG0 and appears as ARG0 2556 in figure 4. The second designated point, D3 488, from figure 1 is replaced by ARG1 and appears as ARG1 2558 in figure 4. The third designated point,

AND 489 and the entire subtree below this designated point (i.e., D4 490 and D0 491) from figure 1 are replaced by ARG2 and appears as ARG2 2559 in figure 4.

The picked point, NAND 485, from figure 1 is now replaced by the name, ADF1, of the new function-defining branch. Thus, ADF1 appears at 2585 in figure 4. Since three points below NAND 485 from figure 1 were designated during the traversal, ADF1 2585 will be given three argument subtrees. The first designated point, D0 486, from figure 1 appears below ADF1 2585 as D0 2586 in figure 4 as the first argument to ADF1 2585. Because D0 is a terminal, D0 2586 appears alone in figure 4. The second designated point, D3 488, from figure 1 appears below ADF1 2585 as D3 2588 in figure 4 as the second argument to ADF1 2585. The third designated point, AND 489, from figure 1 appears below ADF1 2585 as AND 2589 in figure 4 as the third argument to ADF1 2585. Unlike the previous two designated points, AND 489 has a subtree below it in figure 1. The entire subtree is (AND D4 D0) at 489, 490, and 491 in figure 1. Thus, the entire subtree (AND D4 D0) appears as (AND D4 D0) at 2589, 2590, and 2591 of figure 4.

The argument map of the original overall program in figure 1 is {2} and the argument map of the new overall program in figure 4 is {2, 3} because the new branch takes three arguments.

Figures 5 and 1 illustrate an interesting special case of the operation of branch creation.

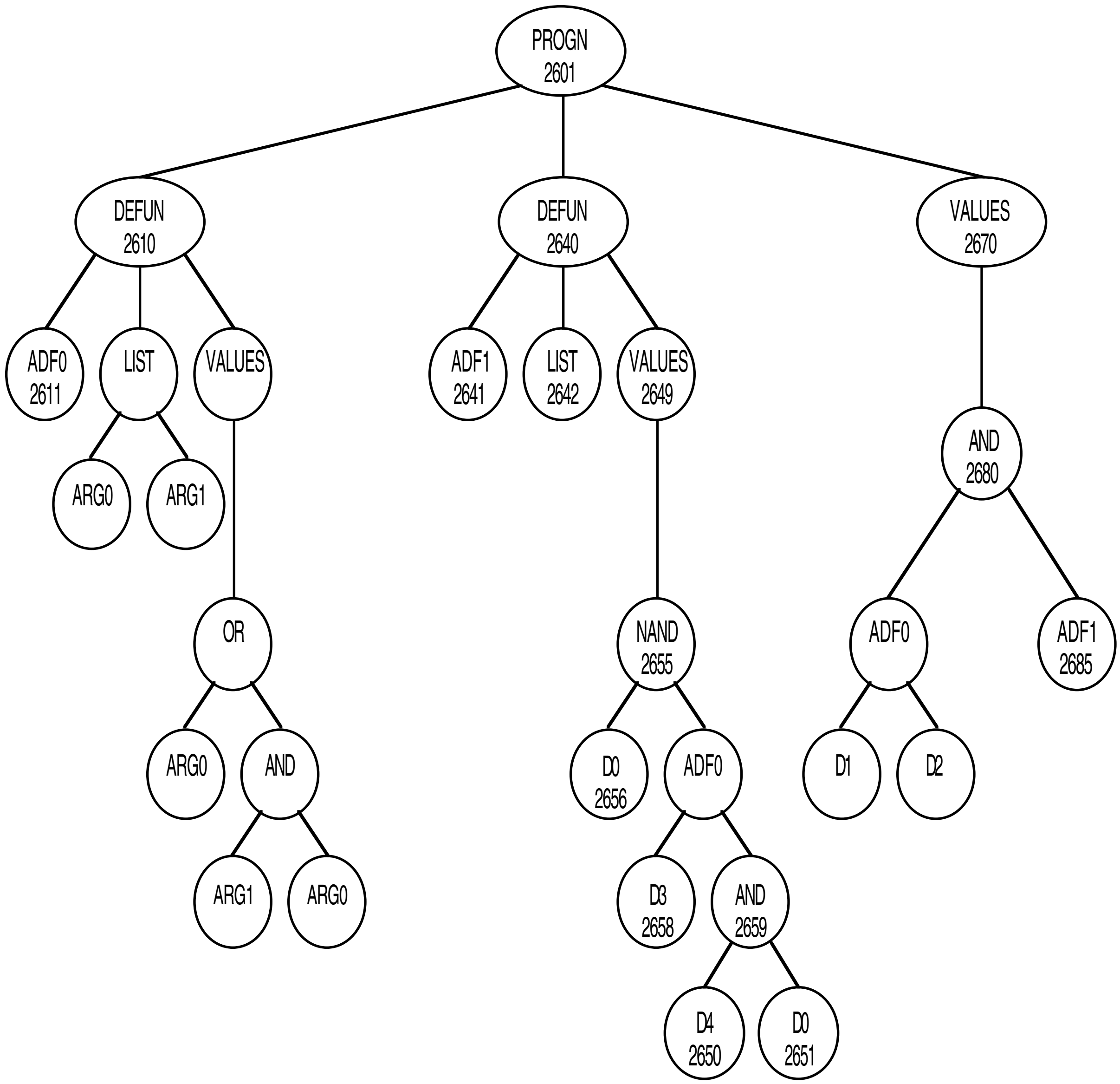

Figure 5: Program with argument map of {2, 0}.

Suppose the point NAND 485 is again picked from the result-producing branch of the program starting at PROGN 400 in figure 1. However, now suppose that, while making the depth-first traversal, no points were designated. As before, the branch creation operation causes a new branch to be added to the overall selected program. The new branch starts at DEFUN 2640 of figure 5. The new branch is given the unique new name, ADF1 (at 2641) in figure 5. However, since no points were designated during the traversal, the argument list of this new branch contains no dummy variables and no dummy arguments appear below LIST 2642 in figure 5. The body of this new branch starts at VALUES 2649 in figure 5. However, since no points were designated during the traversal, the body of the new branch consists of an exact copy of the seven-point subtree starting at the picked point, NAND 485, of figure 1. In particular, several of the actual variables of the problem (D0 2656, D3 2658, D4 2650, and D0 2651) are imported into the body of the function definition for ADF1 in figure 5. As before, the picked point, NAND 485, of figure 1 is replaced in figure 5 by the name, ADF1 2685, of the new function-defining branch. Since no points below NAND 485 were designated during the traversal of figure 1, ADF1 2685 in figure 5 has no

argument subtrees. Thus, in this special case, the entire subtree containing the picked point, NAND 485, and the subtree below it is encapsulated in the zero-argument automatically defined function ADF1 in figure 5. In this example, the argument map of the original overall program in figure 1 is {2} and the argument map of the new overall program in figure 5 is {2, 0}.

The operation of branch creation does not have any immediate effect on the value(s) returned and the action(s) performed by the selected program. However, subsequent operations may alter the branch that is created by this operation and may therefore lead to some later divergence in structure and behavior.

5.6. 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.

Figures 6 and 1 illustrate the operation of argument creation.

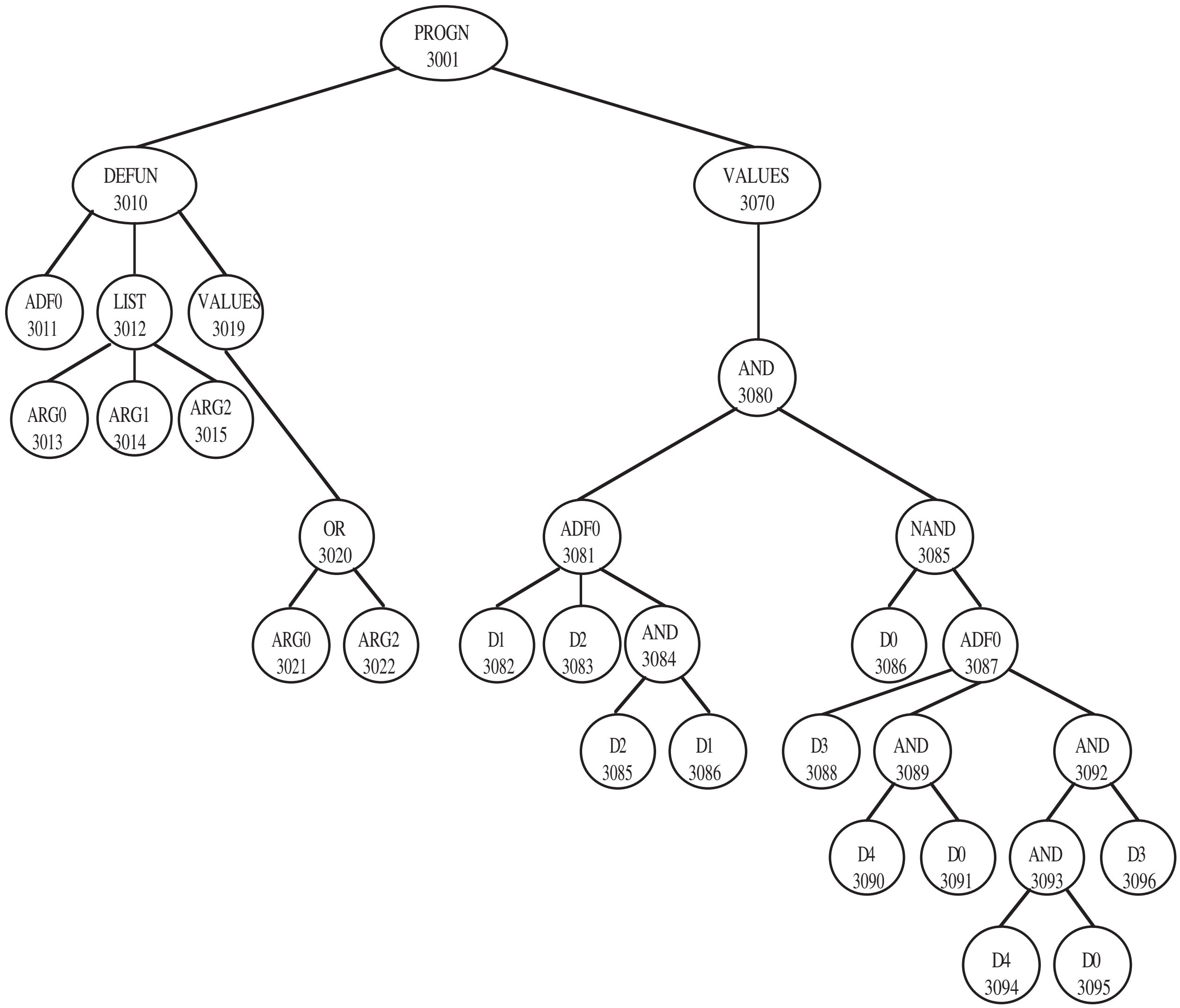

Figure 6: Program with argument map of {3} created using the operation of argument creation.

Suppose the point AND 422 from figure 1 is the picked point, from step (2), in the body of the function-defining branch starting at DEFUN 410. This picked point, AND 422, is the top-most point of the three-point subtree (AND ARG1 ARG0) at 422, 423, and 424. In figure 6, ARG2 is the name given to the newly created argument and ARG2 (labeled 3015) is added to the argument list starting at LIST (labeled 3012) of the picked function-defining branch. In figure 6, ARG2 3022 has replaced the picked point, AND 422, and the entire subtree below the picked point from figure 1.

There are two occurrences of invocations of the picked function-defining branch (ADF0), namely at 481 and 487 in figure 1. The first invocation of ADF0 in the selected program is the two-argument invocation (ADF0 D1 D2) at 481, 482, and 483 in figure 1. A new third argument is added for the invocation ADF0 481. The new argument subtree is manufactured by starting with a copy of the picked point and the entire subtree below that picked point, namely (AND ARG1 ARG0) at 422, 423, and 424 in figure 1. This copy is first modified by replacing the dummy argument ARG1 423 by the argument subtree of the invocation of ADF0 at 481 corresponding to the dummy argument ARG1 423 (i.e., the single point D2 483). Note that it is D2 483 that corresponds because D2 483 is the second argument subtree of ADF0 481 and because

ARG1 is the second dummy argument of DEFUN 410. The copy is further modified by replacing the dummy argument ARG0 424 by the argument subtree of the invocation of ADF0 at 481 corresponding to that dummy argument (i.e., the single point D1 482). The result is that ADF0 is now invoked at 3081 in figure 6 with three (instead of two) arguments, namely

(ADF0 D1 D2 (AND D2 D1)).

The second invocation of ADF0 in the selected program is the two-argument invocation (ADF0 D3 (AND D4 D0)) at 487, 488, 489, 490 and 491 in figure 1. A new third argument is added for the invocation ADF0 487. The new argument subtree is manufactured by starting with a copy of the picked point and the entire subtree below that picked point from figure 1, namely (AND ARG1 ARG0) at 422, 423, and 424. This copy is first modified by replacing the dummy argument ARG1 423 by the argument subtree of the invocation of ADF0 at 487 corresponding to the dummy argument ARG1 423, namely the entire argument subtree (AND D4 D0) at 489, 490, and 491. Note that a three-point subtree, (AND D4 D0), corresponds because it is the second argument subtree of ADF0 487 in figure 1 and because ARG1 is the second dummy argument of DEFUN 410. The copy is further modified by replacing the dummy argument ARG0 424 by the argument subtree of the invocation of ADF0 at 487 corresponding to that dummy argument (i.e., the single point D3 488). The result is that ADF0 is now invoked at 3087 in figure 6 with three (instead of two) arguments, namely

(ADF0 D3 (AND D4 D0) (AND (AND D4 D0) D3)).

The operation of argument creation changes the argument map of the selected program from {2} to {3} because ADF0 now takes three arguments, instead of two.

The operation of argument creation does not have any immediate effect on the value(s) returned and the action(s) performed by the selected program. However, subsequent operations may alter the argument that is created by this operation and may therefore lead to some later divergence in structure and behavior.

6. Rotating the Tires on an Automobile

A *gedanken* experiment can be used to illustrate the role of the six new architecture-altering operations in automated problem-solving. In the experiment, we will visualize four ways of using genetic programming to evolve a procedure to perform the hypothetical task of rotating the tires on an automobile.

The to-be-evolved problem-solving procedure is assumed to have access to all the necessary information from the problem environment (such as the size of the bolts for fastening a tire to an axle, the number of bolts on a tire, the presence or absence of a hubcap, and so forth). In addition, the repertoire of primitive functions includes all the operations necessary to complete the task (such as removing a hubcap, unfastening a bolt, sliding the tire off of the axle, and so forth).

The goal is to evolve a procedure will specify a sequence of operations that will successfully remove each of the four tires and remount them in their specified new locations for any given model of car.

In addition, the to-be-evolved problem-solving procedure will have access to certain extraneous information (such as the color of the car) and certain extraneous operations (such as lifting the hood of the car).

The task is designed to present considerable opportunities for reuse of sequences of procedural steps because all four tires can be removed and remounted in essentially the same way for any particular model of car. The task also presents opportunities for specialization and generalization because the only difference in the procedure for different models of cars lies in what a human planner would consider details (e.g., whether there is a hubcap to remove and remount). In addition, the task is designed to present opportunities for parameterization of procedures (e.g., based on the size and number of the bolts on each tire).

The *gedanken* experiment will visualize the use of genetic programming on this task in the following three ways:

(1) without automatically defined functions,
(2) with automatically defined functions,
(3) with automatically defined functions and with the technique of evolutionary selection of the architecture, and
(4) with automatically defined functions and the six new architecture-altering operations.

6.1. Approach Without Automatically Defined Functions

Five preparatory steps must be performed by the user prior to applying genetic programming without automatically defined functions to a particular problem.

The user must first decide upon the ingredients from which the to-be-evolved programs will be composed. The terminal set for this hypothetical problem consists of the information-carrying variables (e.g., the number and size of the bolts, the presence or absence of hubcaps, the color of the car, etc.). The function set consists of the various primitive operations (e.g., unfastening a bolt, sliding the tire off of the axle, etc.).

The fitness measure of a particular program is computed over a suite of fitness cases consisting of various models of cars of different colors, with and without hubcaps, and with the tires being fastened by different numbers and sizes of bolts. Fitness might be the total number of tires that end up on their respective desired new axles after the complete execution of the program or after some reasonable amount of time.

When automatically defined functions are not being used, the evolved programs, of course, consist only of a result-producing branch.

In principle, genetic programming should be able to evolve a solution to this problem without employing automatically defined functions, provided the population size is sufficiently large and provided that the run is allowed to continue for sufficiently large number of generations.

Although genetic programming usually solves problem in a unexpected ways, we will, for purposes of this *gedanken* experiment, make some statements about the likely characteristics of the evolved solution.

Since automatically defined functions are not being used, each identical or marginally different situation must necessarily be handled by a uniquely-crafted and separately-learned sequence of steps. There is no possiblity of multiple invocation of any part of the evolved code of a program during the execution of the program. Thus, there cannot be one common procedure that is reused four times on each of the tires. Instead, the overall evolved solution will contain four uniquely-crafted (and almost certainly different) sequence of steps for handling each of the four tires. Similarly, there will not be one common procedure that is reused for each of the bolts belonging to each tire. Instead, the solution will contain a uniquely-crafted sequence of steps for unfastening each bolt on each tire.

Moreover, since automatically defined functions are not being used, there would be no possibility for parameterizing the sequence of steps for unfastening the bolts. Thus, models of cars whose tires are fastened by one size of bolts cannot be handled with a common sequence of reusable code in the same way as models with another size of bolts. Instead, there will be one sequence of code that causes the 1/2" wrench to be used when the bolt size is determined to be 1/2" and an additional separate sequence of code that causes the 5/8" wrench to be used on 5/8" bolts. Moreover, there will be one sequence of code for handling models of cars whose tires have six bolts, and a separate sequence of code for handling tires that have five or four bolts.

Because the run without automatically defined functions is incapable of exploiting the considerable underlying symmetry and regularity of this problem environment and incapable of reusing parameterized sequences of code, the overall program necessary to perform this task will be very large. In addition, a large amount of computational effort will have to be expended to evolve this large block of code.

6.2. Approach With Automatically Defined Functions

When automatically defined functions are being used, a solution to a problem employing automatically defined functions can be interpreted such that each automatically defined function represents a subproblem; the body (work) of each automatically defined function represents the solution to a subproblem; and, the result-producing branch represents the assembly of the solution to the subproblems into a solution of the overall problem.

Each automatically defined function also can be interpreted as a change of representation. If, for example, one compares a solution to the overall problem by a result-producing branch that does not invoke any automatically defined functions to one that does, the difference can be viewed as a change of representation. The version of the solution with automatically defined functions solves the problem in terms of the new representation created by the application of the automatically defined functions.

If automatically defined functions are added to genetic programming, we can expect to get a solution to the overall problem that would *reuse* and *parameterize* certain sequences of steps, rather than handling each identical or marginally different situation with a uniquely-crafted and separately-learned sequence of steps.

For example, we can reasonably anticipate a solution in which a tire-dismounting function evolves within the overall multi-part program so that the result-producing branch of the overall program is able to invoke this function with the bolt size as a parameter. When the bolt-unfastening function is parameterized by the bolt size, separate code will not be necessary for handling each different size of bolt.

The ability to generalize is an important aspect of artificial intelligence and automated programming. A parameterized automatically defined function can be interpreted as a generalization in the sense that a parameterized function-defining branch is a generalized procedure for performing some subtask. A particular instantiation of the parameter has the effect of specializing the general procedure to a particular situation.

We can reasonably expect that the size of the overall program will be considerably smaller than the size of the program evolved without automatically defined functions and that less computational effort will have been expended in the evolution of the solution of this problem with automatically defined functions than without them. That is, parsimony should be an emergent consequence of the availability of automatically defined functions.

When automatically defined functions are being used, the evolved program would, of course, consist of a result-producing branch and one or more function-defining branches. The question arises as to how many automatically defined functions will there be? And, how many arguments will each automatically defined function possess?

Before the user can apply genetic programming with automatically defined functions to a problem, the user must employ some technique (probably one of the five existing previously described techniques) for determining the architecture of the overall solution.

Once chosen, the architecture of the overall program constrains the decomposition of the original problem into subproblems. Each particular architecture permits or prohibits certain ways of decomposing the overall problem into subproblems. A particular architecture may facilitate certain decompositions while making others less likely.

The number of available automatically defined functions affects, in general, the character of the solution. If, for example, two or more automatically defined functions are available, we might find separate automatically defined functions devoted to the tire-dismounting and bolt-unfastening subtasks. However, if only one automatically defined function is available, both the tire-dismounting or the bolt-unfastening subtasks would have to be performed inside one automatically defined function or one of these subtasks would have to be performed in the result-producing branch (where there would be no opportunity for parameterized reuse of code).

The number of arguments possessed by each automatically defined functions also generally affects the character of the solution. If, for example, at least one of the available automatically defined functions possesses two or more arguments, then we might see one automatically defined function that is parameterized by several variables. However, if each automatically defined function possesses only one argument, then it would not be possible to take an action inside any of the automatically defined functions based on two or more variables.

6.3. Approach With Evolutionary Selection of the Architecture

As previously mentioned, *evolutionary selection of the architecture* during the run of genetic programming is one of the five existing techniques by which the user can determine the architecture of the overall program (Koza 1994a, chapters 21–25). In

this technique, the population is architecturally diverse starting with the initial random population of generation 0. Then there is a competition among the preexisting architectures in population, However, no architecture is actually created or altered during the run in the technique of evolutionary selection of the architecture.

6.4. Approach With Architecture-Altering Operations

The new architecture-altering operations described herein provide a sixth way to determine the architecture. The architecture-altering operations enable the architecture to be evolved dynamically and automatically during the run of genetic programming in the sense of actually creating new architectures and altering existing architectures during the run.

As previously mentioned, 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.

When automatically defined functions are being used (but without the technique of evolutionary selection of the architecture and without the architecture-altering operations), the third, fourth, and fifth perspectives apply. When automatically defined functions are being used with the technique of evolutionary selection of the architecture (but without the architecture-altering operations), the first perspective applies. Thus, we focus our attention on the second perspective involving specialization and generalization.

As previously mentioned, the initial random population may be uniform and ADF-less; it may be uniform and consist of programs with the minimal ADF structure (i.e., the "minimalist approach"); or, it may be architecturally diverse. We use the "minimalist approach" for purposes of the discussion here.

6.4.1. Generalization by Means of Branch Deletion and Argument Deletion

During a run of genetic programming with the new architecture-altering operations, arguments and branches of programs may be deleted. These deletions may be interpreted as generalizations in the sense that a generalization ignores some previously available information or deletes some previously performed step in carrying out a task. The deletion of an argument or branch may be interpreted as a generalization of the procedure represented by the branch.

Of course, some generalizations are useful and some are not.

The process of generalizing a procedure is often useful because it permits the development of a procedure that is applicable to a wider variety of situations. Generalization may also be desirable because it shortens the description of the procedure (i.e., improves parsimony).

First consider the deletion of an argument.

The deletion of the color of the car from the argument list of a branch would be a helpful simplification of the procedure performed by the branch since there is no possible benefit from this extraneous variable in solving the problem of rotating the tires. If the color of the car is actually used by the branch, deleting this argument will make the branch applicable to a wider variety of situations (with no degradation in performance as measured by the fitness measure of the problem). If the color is ignored, deleting this argument will make the branch more parsimonious.

On the other hand, deletion may make vital information unavailable. Assuming that argument deletion by consolidation or random regeneration are being used, deleting an argument communicating vital information would (almost always) be an unhelpful generalization. The size of the tire's bolts is a necessary argument because the task cannot be performed with a wrench of unsuitable size. The tire-dismounting function can not properly be performed without knowing how many bolts to remove.

Now consider deletion of a branch.

The operation of branch deletion may be viewed as a generalization at the procedural level. After branch deletion by consolidation or random regeneration is performed, the overall program will (usually) be less specialized than before. For example, the deletion of a branch that lifts the car's hood would be a helpful simplification of the procedure performed by the overall program since lifting the hood is not helpful in solving the problem of rotating the tires. On the other hand, it would be unhelpful to delete a branch that checks whether a hubcap is present because the wrench is useless for removing bolts if the bolts are made inaccessible by an unremoved hubcap. In either event, the operation of branch deletion usually produces a procedure with improved parsimony.

There is no way of knowing in advance whether a particular generalization will be helpful. However, natural selection will ultimately judge whether the offspring produced by a particular deletion prove to be more fit or less fit in grappling with the problem environment.

6.4.2. Specialization by Means of Branch Duplication and Argument Duplication

In many instances of automated problem-solving, it is desirable to have the ability to split the problem environment into different cases and treat the cases slightly differently. The operations of branch duplication and argument duplication set the stage for enabling a procedure to be specialized and refined at a later time. Such a specialization or refinement permits such slightly differing treatments of two similar, but different situations. This specialization can occur at the argument level or procedure level.

For example, even though the procedure for changing the tire may be similar for all cars, the procedure must be specialized to each particular type of car by including a certain argument, such as the size of the bolts. Argument duplication provides a way to set the stage so that subsequent evolution can incorporate this additional information.

Branch duplication can provide a way to set the stage so that subsequent evolution can create a slightly different procedure for the two models of cars. The procedure for changing the tire on a Cadillac may differ from the procedure for a Honda

because the Cadillac may have a hubcap that must be removed to expose the bolts so that the wrench can then unfasten the bolts.

During the run, branches or arguments will be duplicated.

For example, consider a program with a branch that hastily tries to unfasten the bolts without checking for the presence of a hubcap. Such a branch works satisfactorily for a car without hubcaps. A program with this branch will accrue a certain amount of fitness for correctly handling the subset of fitness cases involving cars with no hubcaps. However, this program will not receive the maximum possible value of fitness because it will not correctly handle the fitness cases involving cars with hubcaps. After a branch duplication, there are two branches and both start by hastily trying to unfasten the bolts without checking for the presence of a hubcap. Each of these identical branches is invoked under different (randomly chosen) circumstances by their calling branches. Although branch duplication has no immediate semantic effect, subsequent genetic operations may alter one or the other new branch. Some of these alterations will make the overall program more fit at grappling with the problem environment by specializing one branch to some particular situation. For example, one of these branches may be modified by a subsequent genetic operation (e.g., a crossover or mutation) so that it checks for the presence of a hubcap, removes the hubcap, and then continues as before. This specialization is a potential benefit of a branch duplication.

Arguments may also be duplicated during the run. Although argument duplication has no immediate effect, the presence of an additional argument may, after subsequent operations, make the altered program more fit at grappling with certain fitness cases from the problem environment by enabling it to specialize its behavior in response to the additional information provided by the additional argument. Of course, a specialization may be productive, counter-productive, useless or harmless in the context of a particular problem. For example, an argument duplication may make the color of the car available to a branch. This useless information will, at best, be harmless. This argument duplication may be harmful if it overspecializes the branch or reduces the efficiency of operation of the branch.

6.4.3. Specialization by Means of Branch Creation and Argument Creation

When the initial random population is uniformly ADF-less, the operation of branch creation is necessary to create function-defining branches so that part of the initial result-producing branch can become parameterized and generalized. The existence of some parameterized branches are necessary if there are to be multiple invocations of portions of the overall program (and the associated benefit in terms of the overall computational effort necessary to solve the problem). In any event, regardless of how the initial random population is created, the operation of branch creation provides the potential benefit of achieving specialization in the future.

Similarly, the operation of argument creation provides the potential benefit of achieving specialization in the future.

When operating together, the new architecture-altering genetic operations are useful in enabling the simultaneous evolution of the architecture of the overall program for solving a problem while solving the problem. That is, the architecture of the eventual solution to the problem need not be preordained by the user during the preparatory steps. Instead, the architecture can emerge from a competitive fitness-driven process that occurs during the run at the same time as the problem is being solved.

7. 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. The problem requires evolution of a computer program to mimic the behavior of the even-3-parity function.

The frequency of use of the operations of branch duplication, argument duplication, branch deletion, argument deletion, branch creation, and argument creation are controlled by parameters. We often use frequencies in the neighborhood of 1/2% of each operation on each generation; however, for purposes of illustration, we used larger percentages for the two runs described below in order to concentrate more instances of the new architecture-altering operations into a relatively small number of generations. This concentration also enables us to show an entire genealogical audit trail for the first run and the entire maternal lineage for the second run.

7.1. Example 1

The run starts with the random creation of a population of 1,000 individual programs. The "single minimal ADF" approach is used in this example. That is, each program in the initial random population at generation 0 consists of one result-producing branch and a single 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 “single minimal ADF 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).

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

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 on this particular run).

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 since 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.

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

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.

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

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 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 7 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 7. Figure 7 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 7 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 7) 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 7 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.

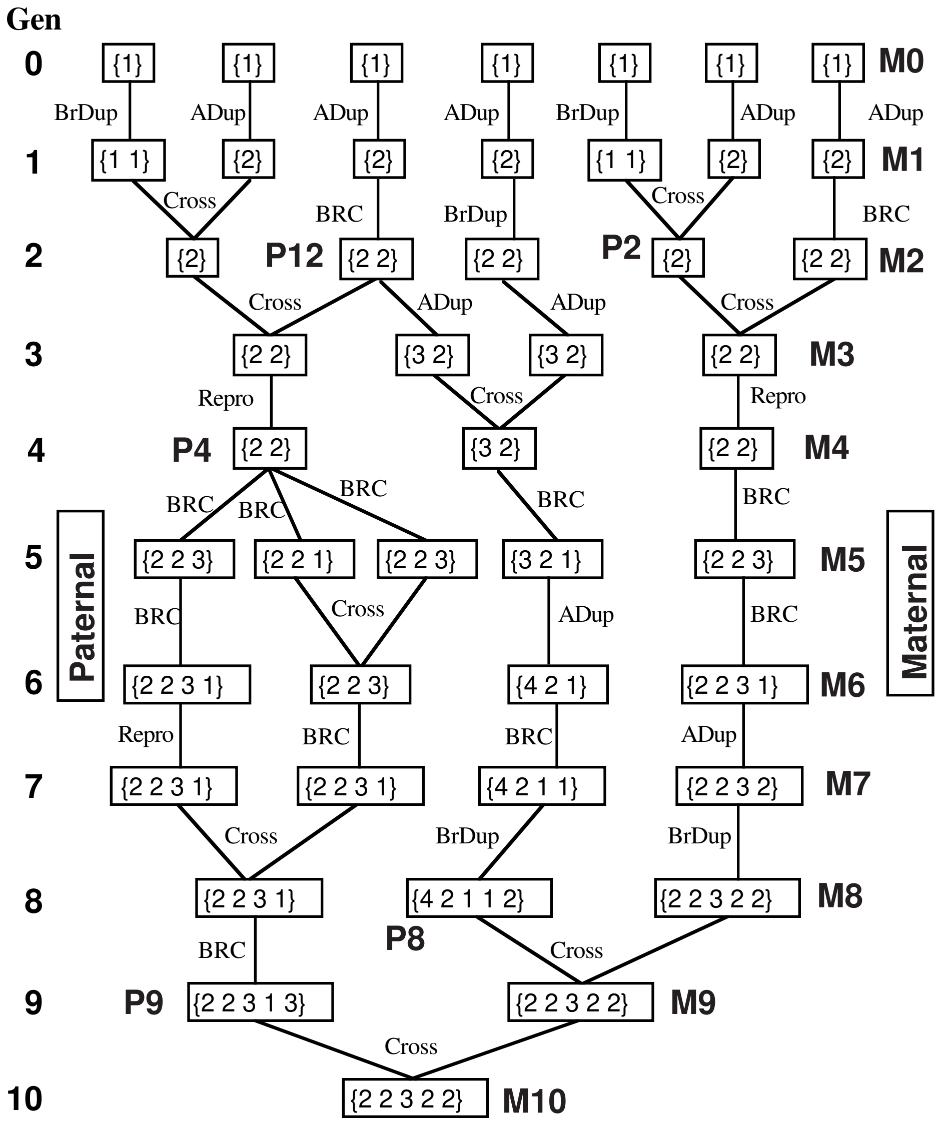

Figure 7: Genealogical audit trail for example 1.

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 7 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 7 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 7 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 7 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 7) 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 7 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 "single minimal ADF" 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.

7.2. Example 2

A second run of the process is now described in order to illustrate the evolution of a hierarchical reference by one function-defining branch of another and to 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.

The best-of-generation program 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 M0 is an early ancestor of the 100%-correct solution that eventually emerges in generation 15. The result-producing branch of this program from generation 0 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).

Figure 8 shows all of the maternal ancestors of the 100%-correct solution from generation 15 (labeled M15 at the bottom of the figure) of the run of example 2. The figure also shows a few selected other ancestors. The figure also shows the argument map of each program in the box for that program. In addition, the raw fitness of each program is shown to the immediate left of its argument map in the box for that program. Because 15 generations are involved, this figure, unlike figure 7, does not show all of the ancestors of the 100%-correct solution, M15, of generation 15. The program labeled M0 at the upper right of this figure is the program from generation 0 shown above, with an argument map of {1} and a raw fitness of 5.

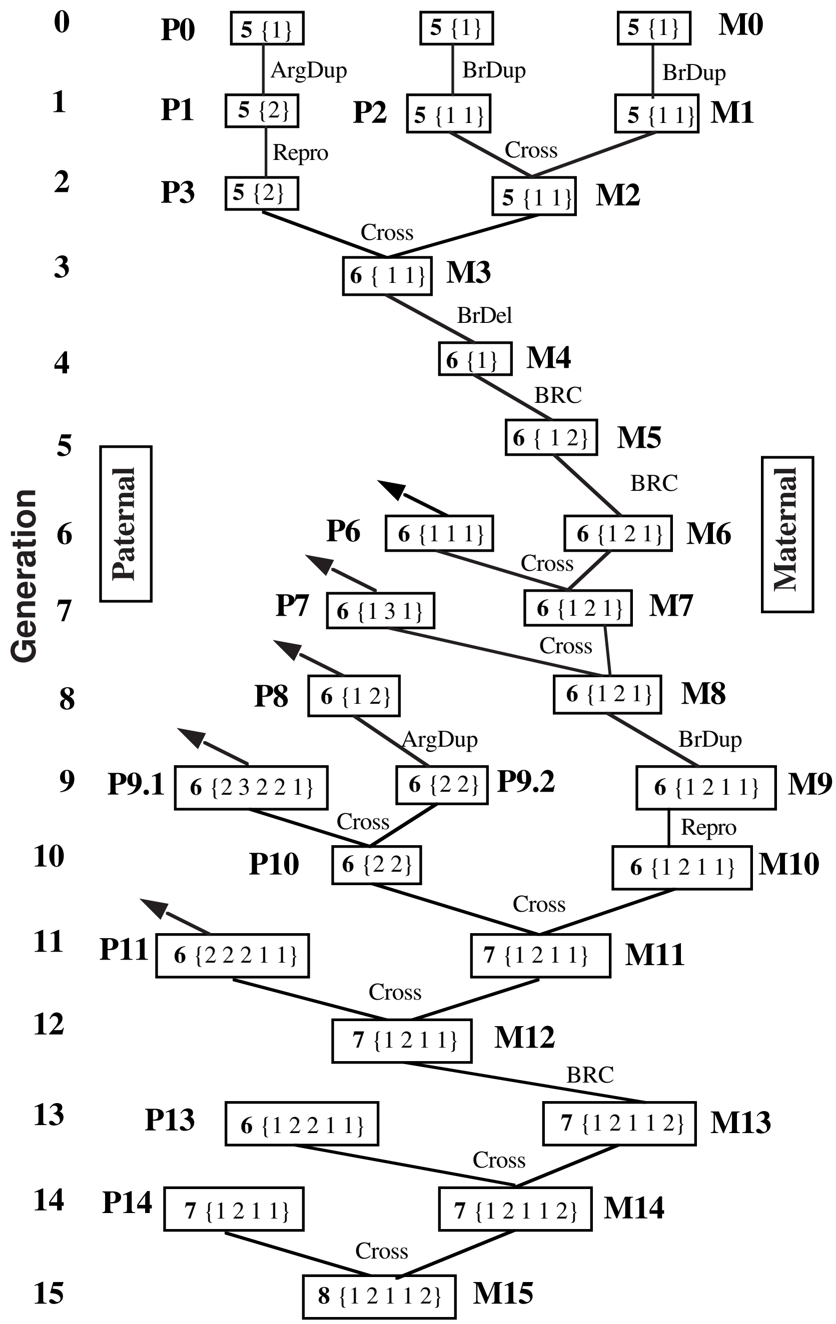

Figure 8: Genealogical audit trail for example 2.

A branch duplication operates on program M0 to produce program M1 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 M3 in generation 3 from 5 to 6.

Program M3 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 M3 is the exactly the same as ADF0 of ancestor M0 at generation 0 (i.e., the NOT function). The branch duplication that created program M1 in generation 1 duplicated ADF0 of ancestor M0 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 M3 to produce program M4 in generation 4. The surviving function-defining branch of program M4 is exactly the same as ADF0 of ancestor M0 at generation 0 (i.e., it is the NOT function). Program M4 retains a fitness level of 6.

The result-producing branch of program M4 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 M5 of generation 5 from the underlined portion of the result-producing branch of program M4 above. The new branch, ADF1, is shown below:

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

The result-producing branch of program M4 of generation 4 is also modified and the modified version is part of program M5 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 M10 is the NOT function. ADF1 of program M10 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 M10 mates with father P10 to produce offspring M11 in generation 11.

ADF0 of father P10 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 P10 was inserted into ADF1 of mother M10 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))))).
```

As can be seen, a hierarchy of function definitions has emerged to solve the problem of symbolic regression of the even-3-parity problem using the new architecture-altering operations.

8. Conclusions

We have shown that it is possible to evolve the architecture using the architecture-altering operations while simultaneously solving a problem.

9. Future Work

More computational effort is required to perform both tasks simultaneously than to merely solve the problem when the architecture is given and fixed. Future work will attempt to quantify this amount of additional computation effort.

10. Acknowledgements

David Andre and Walter Alden Tackett wrote the computer program in ANSI C for an IBM PC type of computer to implement five of the six architecture-altering operations described above. Simon Handley made numerous helpful comments on this paper.

11. Bibliography

Axelrod, Robert. *The Evolution of Cooperation*. New York: Basic Books 1984.
Axelrod, Robert. The evolution of strategies in the iterated prisoner's dilemma. In Davis, Lawrence (editor). *Genetic Algorithms and Simulated Annealing* London: Pittman 1987.
Angeline, P. J. 1994a. *Evolutionary Algorithms and Emergent Intelligence*. Ph.D. dissertation. Computer Science Department. The Ohio State University.
Angeline, P. J. 1994b. Genetic programming and the emergence of intelligence. In Kinnear, K. E. Jr. (editor). *Advances in Genetic Programming*. The MIT Press.
Angeline, Peter J. and Pollack, Jordan B. 1993. Competitive environments evolve better solutions for complex tasks. In Forrest, Stephanie (editor). *Proceedings of the Fifth International Conference on Genetic Algorithms*. San Mateo, CA: Morgan Kaufmann Publishers Inc. Pages 264-270.
Angeline, Peter J. and Pollack, Jordan B. 1994. 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.
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.
Davis, L. (editor). 1987. Genetic Algorithms and Simulated Annealing. Pittman.
Davis, L. 1991. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold.
Dyson, Paul and Sherratt, David. 1985. Molecular mechanisms of duplication, deletion, and transposition of DNA. In Cavalier-Smith, T. (editor). *The Evolution of Genome Size*. Chichester: John Wiley & Sons.
Forrest, S. (editor). 1993. *Proceedings of the Fifth International Conference on Genetic Algorithms*. Morgan Kaufmann.
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*. Volume 231. Pages 324–334.
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.
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. 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., and Rice, James P. 1992 .*Genetic Programming: The Movie*. Cambridge, MA: The MIT Press.

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*. Volume 20. Pages 81-108.

Michalewicz, Z. 1992. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag.

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*. Volume 1. Pages 351–361.

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

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

Computer Science Department October 21, 1994

ARCHITECTURE-ALTERING OPERATIONS FOR EVOLVING THE ARCHITECTURE OF A MULTI-PART PROGRAM IN GENETIC PROGRAMMING

John R. Koza

Computer Science Department
Stanford University
Margaret Jacks Hall
Stanford, California 94305–2140 USA

ARCHITECTURE-ALTERING OPERATIONS FOR EVOLVING THE ARCHITECTURE OF A MULTI-PART PROGRAM IN GENETIC PROGRAMMING

John R. Koza
Computer Science Department
Stanford University
Margaret Jacks Hall
Stanford, California 94305-2140 USA
Koza@CS.Stanford.Edu
Phone: 415-941-0336
FAX: 415-941-9430

ABSTRACT

Previous work described a way to evolutionarily *select* the architecture of a multi-part computer program from among preexisting alternatives in the population while concurrently solving a problem during a run of genetic programming. This report describes six new architecture-altering operations that provide a way to *evolve* the architecture of a multi-part program in the sense of *actually changing* the architecture of programs dynamically during the run.

The new architecture-altering operations are motivated by the naturally occurring operation of gene duplication as described in Susumu Ohno's provocative 1970 book *Evolution by Means of Gene Duplication* as well as the naturally occurring operation of gene deletion.

The six new architecture-altering operations are branch duplication, argument duplication, branch creation, argument creation, branch deletion and argument deletion.

A connection is made between genetic programming and other techniques of automated problem solving by interpreting the architecture-altering operations as providing an automated way to specialize and generalize programs.

The report demonstrates that a hierarchical architecture can be evolved to solve an illustrative symbolic regression problem using the architecture-altering operations.

Future work will study the amount of additional computational effort required to employ the architecture-altering operations.