The Evolution of Evolvability in Genetic Programming [1]

Lee Altenberg

Institute of Statistics and Decision Sciences, Duke University
Durham, NC 27708-0251 Internet: altenber@acpub.duke.edu

The notion of "evolvability" — the ability of a population to produce variants fitter than any yet existing — is developed as it applies to genetic algorithms. A theoretical analysis of the dynamics of genetic programming predicts the existence of a novel, emergent selection phenomenon: the evolution of evolvability. This is produced by the proliferation, within programs, of blocks of code that have a higher chance of increasing fitness when added to programs. Selection can then come to mold the *variational* aspects of the way evolved programs are represented. A model of code proliferation within programs is analyzed to illustrate this effect. The mathematical and conceptual framework includes: the definition of evolvability as a measure of performance for genetic algorithms; application of Price's *Covariance and Selection Theorem* to show how the fitness function, representation, and genetic operators must interact to produce evolvability — namely, that genetic operators produce offspring with fitnesses specifically correlated with their parent's fitnesses; how blocks of code emerge as a new level of replicator, proliferating as a function of their "constructional fitness", which is distinct from their schema fitness; and how programs may change from innovative code to conservative code as the populations mature. Several new selection techniques and genetic operators are proposed in order to give better control over the evolution of evolvability and improved evolutionary performance.

3.1 Introduction

The choice of genetic operators and representations has proven critical to the performance of genetic algorithms (GAs), because they comprise dual aspects of the same process: the creation of new elements of the search space from old. One hope in genetic algorithm research has been that the representation/operator problem could itself be solved through an evolutionary approach, by creating GAs in which the representations and/or operators can themselves evolve. What I discuss in this chapter is how genetic programming (GP) exhibits the evolution of representations as an inherent property. Moreover, I investigate how the direction of evolution of representations in genetic programming may be toward increasing the *evolvability* of the programs that evolve, and suggest ways that the evolution of evolvability can be controlled.

[1] Chapter 3 in *Advances in Genetic Programming*, K. E. Kinnear, Jr., ed. pp. 47-74.

3.1.1 Evolvability

"Evolvability" is a concept I wish to develop as a performance measure for genetic algorithms. By evolvability I mean the ability of the genetic operator/representation scheme to produce offspring that are fitter than their parents. Clearly this is necessary for adaptation to progress. Because adaptation depends not only on how *often* offspring are fitter than their parents, but on how *much* fitter they are, the property ultimately under consideration, when speaking of evolvability, is the entire distribution of fitnesses among the offspring produced by a population. Since there is a chance that offspring are fitter than parents even in random search, good GA performance requires that the upper tail of the offspring fitness distribution be fatter than that for random search.

But this excess of fitter offspring as compared with random search needn't occur with all parents. It need only occur with parents that are fitter than average, because selection is biasing the population in that direction. In other words, the action of the genetic operator on the representation needs to produce high correlations between the performance of parents and the fitness distribution of their offspring. This is shown in Section 3.2.3 through an application of Price's *Covariance and Selection Theorem* [Price 1970]. Price's Theorem in my mind serves as a fundamental theorem for genetic algorithms.

This correlation property is exemplified by the Building Blocks hypothesis [Goldberg 1989]. A building block is a schema [Holland 1975] from a fitter-than-average individual which, when recombined into another individual, is likely to increase its fitness. Thus a building block is *defined* by a correlation between parent and offspring fitnesses under recombination. A recombination operator which is able to pick out this schema from one individual and substitute it into another is then leaving intact the building blocks but producing variation in a way for which there is adaptive opportunity, namely, combining the proper building blocks. But for recombination to be of use, however, the representation must be such that building blocks — i.e. schemata which give correlated fitnesses when substituted into a chromosome by recombination — exist.

Evolvability can be understood to be the most "local" or finest-grained scale for measuring performance in a genetic algorithm, whereas the production of fitter offspring over the course of a run of a GA, or many runs, are more "global", large scale performance measures. As a population evolves, the distribution of offspring fitnesses is likely to change. The global performance of a genetic algorithm depends on it maintaining the evolvability of the population as the population evolves toward the global optimum. I will explore how genetic programming, through its ability to evolve its representations, may be able to maintain or increase the evolvability of the programs as a population evolves.

3.1.2 Representations

How is evolvability achieved in genetic algorithms? It comes from the genetic operators being able to transform the representation in ways that leave intact those aspects of the individual that are already adapted, while perturbing those aspects which are not yet highly adapted. Variation should be channeled toward those "dimensions" for which there is selective opportunity.

As an illustrative domestic example, consider the ways that water flow and temperature in a bathroom shower are "represented". There are two common control mechanisms:

1. One handle for hot water flow and one handle for cold water flow; and

2. A single handle which varies water flow by tilting up and down, and varies temperature by twisting clockwise and counterclockwise.

Adjusting the shower consists of perturbing the two degrees of freedom in either representation until flow and temperature are satisfactory. Representation 2 was invented to deal with the problem that 1 has, which is that bathers usually want to change the flow rate without changing the temperature, or vice versa. Under 1, that would require a sophisticated "operator" which changes both handles simultaneously in the right amounts, while in 2 it requires changing only the tilt of the handle or the twist. It generally takes many more adjustments under 1 to converge to satisfaction, with greater risks of scalding or freezing. The representation under 2 is more "attuned" to the dimensions of variation typical of the environment [Barwise and Perry 1983]. Another way of saying this is that the two degrees of freedom in 1 are more epistatic in their relation to the objective function of the bather.

The basic design task for genetic algorithms is to create a system whose "dimensions of variability" match the dimensions of variability of the task domain. The existence of sub-tasks or sub-goals in a problem correspond to dimensions of variability for the task domain. Representations and operators that are attuned to these dimensions can therefore exhibit greater evolvability.

When speaking about the evolution of representations in genetic programming, I mean not the fine-scale aspects of a genetic program, such as the choice of functions or terminals, the data-structure implementation, or LISP versus some other programming language; these are fixed at the outset. Rather, I mean the large-scale structure of the program.

Any particular program is a representation of the *behavior* that the program exhibits. Often the same program behavior can be represented by a variety of programs. These programs may exhibit different *evolvability*, however, because some representations may make it more likely that modification by genetic operators can produce still fitter programs. A representation must therefore be understood to have *variational* properties — how changes

in the representation map to changes in behavior of the code. Object oriented programming, for example — although not usually described in this manner — is fundamentally a method of engineering desirable variational properties into code. So it is important to understand that the variational properties of a program are distinct from its fitness. In classical fixed-length GAs, selection can act only on the fitness of the population, not on the variational properties of the representation. In genetic programming, selection can act on both, as will be described.

3.1.3 Evolving evolvability

A number of modifications of genetic algorithms have been implemented to evolve their evolvability by adapting the genetic operators or representations. These include:

• the addition of modifier genes, whose allelic values control the genetic operators (e.g. loci controlling recombination or mutation rates and distributions) [Bäck, Hoffmeister and Schwefel 1991, Bergman and Feldman 1992, Shaffer and Morishima 1987];

• running a meta-level GA on parameters of the operators, in which each run of the primary GA produces a performance measure used as the fitness of the operator parameter [Grefenstette 1986];

• "messy GAs", a form of representation which allows more complex transformations from parent to offspring [Goldberg, Deb, and Korb 1990];

• focusing the probabilities of operator events to those that have had a history of generating fitter individuals [Davis 1989];

• adjusting the mutation distribution to maintain certain properties of the fitness distribution, e.g. the "one-fifth" rule of Evolution Strategies [Bäck, Hoffmeister and Schwefel 1991];

• dynamic parameter encoding [O'Neil and Shaefer 1989, Pitney, Smith, and Greenwood 1990, Schraudolph and Belew 1992, Shaefer 1987, Szarkowicz 1991].

Genetic programming allows another means by which evolvability can evolve: the differential proliferation of blocks of code *within* programs. Such differential proliferation of blocks of code within programs has already been observed in practice (see [Angeline and Pollack 1992, Tackett 1993], and the chapters in this volume by Angeline (Ch. 4), Kinnear (Ch. 6), Teller (Ch. 9), Oakley (Ch. 17), and Handley (Ch. 18)).

3.1.4 Constructional selection

Proliferation of copies of code within programs is possible in genetic programming because of the syntactic closure of programs to such duplications. The recombination operators

carry out these duplications. The exchange of whole subtrees has been the main genetic operator used in GP, but in general, recombination can involve any block of code (e.g. Automatically Defined Functions [Koza 1993]).

Because blocks of code can be duplicated indefinitely within programs, they constitute an emergent level of replication, and hence, selection. Here I employ the concept of "replicator": any entity that can spawn multiple copies of itself based on its interactions with an environment [Dawkins 1976, Brandon 1988]. Programs are the primary level of replicator in GP, since the selection operator explicitly duplicates them based on their fitness. Blocks of code can emerge as another level of replicator because copies of them can proliferate within programs. The rate at which they proliferate is not imposed by the selection operator as in the case of the programs themselves.

Suppose that different blocks of code had different likelihoods of improving fitness when substituted at random locations in randomly chosen programs in the population. If the genetic operator could increase the frequency with which it chooses those blocks of code bearing the highest likelihood of producing fitter offspring, then evolvability could increase. This frequency depends on the average number of copies of that block of code in the population, which can increase in two ways:

The first way is for programs that contain the block to increase their number in the population. This shows up in the "schema" or marginal fitness of a block of code, and is accounted for by applying the Schema Theorem [Holland 1975] to genetic programming [Koza 1992]. But this pure selection effect does not produce more copies of a block of code within programs. That requires the creation of novel programs through insertion of additional copies of the code from donor programs to recipient programs. If the offspring programs survive long enough, they will be recipients of furhter copies of the code, or will be used as donors for contributing additional copies of the code to other recipient programs.

There are thus two numbers that can evolve: the number of programs carrying a block of code, and the number of copies of the code within the programs. Change in the former is determined by selection as specified by the fitness function of the GP algorithm. But change in the latter is an emergent selection phenomenon, involving the effects of constructing new programs by adding copies of the block of code. What I am proposing is that the proliferation of copies of code within programs constitutes a secondary kind of selection process, which I will refer to as "constructional selection" [Altenberg 1985]. The Schema Theorem does not account for this latter process.

Increases in the number of copies of a block of code within programs requires multiple sampling by the genetic operator; it can't be the result of a one-time lucky event. Therefore, differential proliferation of code depends on repeatable *statistical* differences in the fitness effects of code substitutions by the different blocks. (In contrast, in a fixed-length GA, a single substitution of a schema into a chromosome can produce an individual so fit that

the schema goes to fixation without further recombination events; at that point no further increase in the schema frequency is possible).

If a block of code has a relatively stationary distribution in its effects on the fitness of programs to which it is added, its rate of proliferation *within* programs can be referred to as its "constructional" fitness. The constructional fitness is distinct from the current average fitness in the population of a block of code (i.e. its marginal or "schema" fitness). It is an emergent selection phenomenon in that the constructional fitness is never evaluated by the GP algorithm, nor is it obtained by averaging current program fitnesses.

What determines the constructional fitness of a block of code? That is, how does the fitness distribution, for recombinants that carry a new copy of a block of code, determine whether that block proliferates or not? How rapidly a particular block of code proliferates within programs depends in nontrivial ways on the relationship between the population dynamics, the fitness distribution of the new programs carrying additional copies, and how stationary this distribution is as the population evolves.

The population dynamics which are most easy to analyze would be a GP version of "hill-climbing". A single program is modified by internally duplicating a block of code and substituting it at another location within the program. If the new program is fitter, keep it; if not, try again. GP hill-climbing will be the effective dynamics if very low recombination rates are used so that selection fixes the fittest program before the next recombination event occurs. GP hill-climbing resembles the dynamics of gene duplication in natural populations.

Under these dynamics, deleterious additions of a block of code go extinct before the code can be passed on again through recombination. Only advantageous additions are ever resampled. Therefore blocks of code proliferate based solely on their probability of increasing fitness when added to a program. As a consequence, in time, a growing proportion of code additions made by the genetic operator should involve blocks of code with a high probability of producing a fitness increase. In this way, the operator becomes focused in its action on those dimensions of variability with greater potential for increasing fitness. This is not simply an increase in the adaptedness of the programs in the population, but a change in their potential for further adaptation, i.e. a change in the evolvability of the gene pool.

Under the high recombination rates typical of GP, the picture is not so simple. Let me refer to a block of code's probability of increasing fitness when added to a program as its "evolvability value". With high recombination rates, blocks of code with the highest evolvability value need not have the highest constructional fitness. Even a program on its way to extinction under selection can be sampled several times by the genetic operator as a donor of its code to new programs. Thus a block of code that caused a deleterious effect

when added to a program has a chance of being sampled again for further donation before its program goes extinct.

Blocks of code will proliferate depending on the survival time of the programs resulting from their addition. So code that produces a near neutral effect could have a constructional fitness advantage over "Babe Ruth"[2] code — i.e. code that occasionally produces very fit new programs, but which on the average has a deleterious effect when added to a program.

In the early stages of the evolution of a population, for a block of code to survive (distinct from the question of proliferation within programs), it must be able to increase its fitness through recombination events with other blocks of code at least as fast as the mean fitness of the population is increasing. Code is then in competition for high evolvability, and so one would expect code with high evolvability to survive and/or proliferate within programs at that stage.

In the later stages of the population's evolution, the population reaches a recombination-selection balance, and the mean fitness approaches a quasi-stationary state. At that point the main selective opportunity is for a fitter-than-average program to preserve what fitness it's got when producing offspring [Altenberg 1984, Altenberg and Feldman 1987]. Code is then in competition for the robustness of its behavior in the face of the actions of the genetic operators.

The evolution of populations of programs whose fitness is robust to crossover in the later stages of GP runs has been reported [Koza 1992], and Andrew Singleton and Mick Keenan (personal communication) have identified this phenomenon as "defense against crossover".

The possibilities in genetic programming for such emergent evolutionary phenomena indicate that it may prove to be a rich area for research. But it shows that a better understanding of GP dynamics is needed to control the direction of such emergent phenomena toward making GP a useful tool. Some suggestions are provided in the Discussion for how to maintain the evolution of evolvability through special selection regimes and genetic operators.

3.1.5 Synopsis of the models

In the next two sections, I provide some mathematical underpinnings to the concepts discussed so far. In section 3.2 I investigate the relations between selection, genetic operators, representations and evolvability. The main results are that, first, there is a duality between representations and genetic operators in producing a *transmission function*; second, it is the relationship between the transmission function and the selection function that determines the performance of the genetic algorithm; and third, the relationship between selection and

[2] Babe Ruth, the legendary American baseball player, had a mediocre batting average, but when he got a hit, it was very likely to be a home run.

transmission which produces good GA performance is that there be correlations between parental fitness and offspring fitness distributions. This is expressed mathematically by elaborating on Price's Covariance and Selection Theorem [Price 1970]. I propose that this theorem provides a more adequate basis to account for genetic algorithm performance than does the Schema Theorem [Holland 1975].

In section 3.3, I offer a simple model of genetic programming dynamics to illustrate how constructional selection can produce the evolution of evolvability. The model includes simplifying assumptions made to provide a tractable illustration of the mechanism of evolutionary increase in evolvability, but these simplifications undoubtedly limit the model's applicability to other emergent phenomena in genetic programming. These assumptions include that the genetic operator can add blocks of code to programs from other programs, but cannot delete blocks of code from programs or disrupt them, and that the population undergoes GP hill-climbing. But this model comprises a "generation 0" model from which other more complex models are being developed. It demonstrates how the different distributions of fitness effects of blocks of code lead to the exponential proliferation of those which confer greater evolvability to the programs that contain them. I also emphasize the distinction between the schema fitness of a block of code and its "constructional" fitness, which is what determines its proliferation rate within programs.

3.2 Selection, Transmission, and Evolvability

The conventional theoretical approach to genetic algorithms focuses on schemata and the Schema Theorem. This approach, however, relies on hidden assumptions which have obscured and even misled GA design (e.g. the principle of minimal alphabets [Goldberg 1989, Antonisse 1989]). The main hidden assumption I wish to bring out is that of "implicit correlations". This is approached by considering a general model of genetic algorithms with arbitrary representations, operators, and selection. With this approach, it is possible to extract features that are essential for the performance of GAs which have nothing to do with schemata.

The action of genetic operators on the representation produces a *transmission function* [Slatkin 1970, Cavalli-Sforza and Feldman 1976, Altenberg 1984, Altenberg and Feldman 1987], which is simply the probability distribution of offspring from every possible mating. With two parents, for example, the transmission function would simply be

$$T(i \leftarrow j, k),$$

where j and k are labels for the parents and i the label for an offspring. To be precise, let $\mathcal{S}$ be the search space; then $T : \mathcal{S}^3 \mapsto [0, 1]$, $T(i \leftarrow j, k) = T(i \leftarrow k, j)$, and $\sum_i T(i \leftarrow j, k) = 1$ for all $i, j, k \in \mathcal{S}$.

Clearly, the transmission function can represent not only the action of genetic operators in binary, fixed length GAs, but also messy GAs, the parse trees of GP, and real-valued GAs as well (where $\mathcal{S} \subset \Re^n$) or even functionals. Multi-parent transmission could simply be represented as $T(i \leftarrow j_1, \ldots, j_m)$.

Principle 1 *It is the relationship between the transmission function and the fitness function that determines GA performance. The transmission function "screens off" [Salmon 1971, Brandon 1990] the effect of the choice of representation and operators, in that changes in either affect the dynamics of the GA only through their effect on the transmission function.*

For example, the effect of inserting "introns" (loci that are neutral to performance but are targets of crossover) [Levenick 1991, Forrest and Mitchell 1993], which is a change in the representation, can equivalently be achieved by an adjustment in the crossover operator. In the Discussion some implications of Principle 1 for genetic programming design will be considered.

What aspect of the relationship between transmission function and fitness function is crucial to GA performance? It is the correlation between parental performance and offspring performance under the action of the genetic operators. Without such a correlation, selection on the parents has no means of influencing the distribution of offspring fitnesses. I make explicit this condition in Section 3.2.3, where I utilize Price's *Selection and Covariance* theorem [Price 1970, 1972] to show how the change in the fitness distribution of the population depends on this correlation.

To develop this I first present the form of the recursion for the general "canonical genetic algorithm". Then I define the idea of "measurement functions" to extract information from a population, which allows the statement of Price's theorem. Use of the proper measurement function allows one to use Price's theorem to show how the distribution of fitnesses in the population changes over one generation, including the probability of producing individuals fitter than any in the population. This will be seen to depend on the covariance between parent and offspring fitness distributions, and a "search bias" indicating how much better in the current population the genetic operator is than random search.

3.2.1 A general model of the canonical genetic algorithm

For the purpose of mathematical exposition, a "canonical" model of genetic algorithms has been generally used since its formulation by Holland [1975], which incorporates

assumptions common to many evolutionary models in population genetics: discrete, non-overlapping generations, frequency-independent selection, and infinite population size. The algorithm iterates three steps: selection, random mating, and production of offspring to constitute the population in the next generation.

Definition: Canonical Genetic Algorithm
The dynamical system representing the "canonical" genetic algorithm is:

$$x_i' = \sum_{jk} T(i \leftarrow j,k)\, \frac{w_j w_k}{\overline{w}^2}\, x_j x_k, \tag{3.1}$$

where

- x_i *is the frequency of type* i *in the population,* $i = 1 \ldots n$*, and* x_i' *is the frequency in the next generation;*
- w_i *is the fitness of type* i*;* [3]
- $\overline{w} = \sum_i w_i x_i$ *is the mean fitness of the population; and*
- $T(i \leftarrow j,k)$ *is the probability that offspring type* i *is produced by parental types* j *and* k *as a result of the action of genetic operators on the representation.*

This general form of the transmission-selection recursion was used at least as early as 1970 by Slatkin [1970], and has been used subsequently for a variety of quantitative genetic and complex transmission systems [Cavalli-Sforza and Feldman 1976, Karlin 1979, Altenberg and Feldman 1987]..

The quadratic structure of the recursion can be seen by displaying it in vector form as:

$$\boldsymbol{x}' = \frac{1}{\overline{w}^2}\, \boldsymbol{T}\, (\boldsymbol{W} \otimes \boldsymbol{W})\, (\boldsymbol{x} \otimes \boldsymbol{x}) \tag{3.2}$$

where

$$\boldsymbol{T} = \parallel T(i \leftarrow j_1, j_2) \parallel_{i,j_1,j_2=1}^{n}, \tag{3.3}$$

is the *transmission matrix*, the n by n^2 matrix of transmission function probabilities, n is the number of possible types (assuming n is finite), $\boldsymbol{W}$ is the diagonal matrix of fitness values, $\boldsymbol{x}$ is the frequency vector of the different types in the population, and $\otimes$ is the Kronecker (tensor) product. The Kronecker product gives all products of pairs of matrix

[3] Using the letter "w" for fitness is traditional in population genetics, and derives from "worth" (Sewall Wright, informal comment).

elements, e.g.:

$$\| \, x \;\; y \;\; z \, \| \otimes \left\| \begin{array}{cc} a & b \\ c & d \\ e & f \end{array} \right\| = \left\| \begin{array}{cccccc} xa & xb & ya & yb & za & zb \\ xc & xd & yc & yd & zc & zd \\ xe & xf & ye & yf & ze & zf \end{array} \right\|.$$

The mathematics of (3.2) for $\boldsymbol{W} = \boldsymbol{I}$ (the identity matrix) have been explored in depth by Lyubich [1992].

3.2.2 Measurement functions

Let $F_i : \mathcal{S} \mapsto \mathcal{V}$ be a function of the properties of individuals of type $i \in \mathcal{S}$ (or let $F_i(p) : \mathcal{S} \mapsto \mathcal{V}$ be a parameterized family of such functions, for some parameter p). Here we are interested in properties that can be averaged over the population, so $\mathcal{V}$ is a vector space over the real numbers (e.g. $\Re^n$ or $[0,1]^n$), allowing averages $\overline{F} = \sum_i x_i \, F_i \in \mathcal{V}$ for $x_i \geq 0$ with $\sum_i x_i = 1$.

Different measurement functions allow one to extract different kinds of information from the population, by examining the change in the population mean for the measurement function:

$$\overline{F} = \sum_i F_i \; x_i, \quad \overline{F}' = \sum_i F_i \; x_i' \tag{3.4}$$

Some examples of different measurement functions and the population properties measured by $\overline{F}$ are shown in Table 3.1.

3.2.3 Price's theorem applied to evolvability

Price's theorem gives the one-generation change in the population mean value of F:

Theorem 1 (Covariance and Selection, Price, 1970)
For any parental pair $\{j,k\}$, let ϕ_{jk} represent the expected value of F among their offspring. Thus:

$$\phi_{jk} = \sum_i F_i \; T(i \leftarrow j,k).$$

Then the population average of the measurement function in the next generation is

$$\overline{F}' = \overline{\phi}_u + \mathrm{Cov}[\phi_{jk}, \; w_j w_k / \overline{w}^2] \tag{3.5}$$

where:

$$\overline{\phi}_u = \sum_{jk} \phi_{jk} \; x_j x_k$$

Table 3.1
Measurement functions, F_i (some taking arguments), and the population properties measured by their mean $\overline{F}$.

Population Property:	Measurement Function:
• Mean fitness:	$F_i = w_i$
• Fitness distribution:	$F_i(w) = \begin{cases} 1 & w_i \le w \\ 0 & w_i > w \end{cases}$
• Subtree frequency:	$F_i(\mathcal{S}) = \Pr[\text{Operator picks subtree } \mathcal{S} \text{ for crossover}]$
• Schema $\mathcal{H}$ frequency:	$F_i(\mathcal{H}) = \begin{cases} 0 & i \notin \mathcal{H} \\ 1 & i \in \mathcal{H} \end{cases}$
• Mean phenotype:	$F_i \in \Re^n = \{\text{vector-valued phenotypes}\}$

is the average offspring value in a population reproducing without selection, and

$$\mathrm{Cov}[\phi_{jk},\ w_j w_k/\overline{w}^2] = \sum_{jk} \phi_{jk}\, \frac{w_j w_k}{\overline{w}^2}\, x_j x_k - \overline{\phi}_u$$

is the population covariance between the parental fitness values and the measured values of their offspring.

Proof. This is obtained directly from substituting (3.1) into (3.4). ■

Price's theorem shows that the covariance between parental fitness and offspring traits is the means by which selection directs the evolution of the population.

Now, Price's theorem can be used to extract the change in the *distribution* of fitness values in the population by using the measurement function

$$F_i(w) = \begin{cases} 0 & w_i \le w \\ 1 & w_i > w \end{cases}.$$

Then

$$\overline{F}(w) = \sum_i F_i(w)\, x_i = \sum_{i:w_i > w} x_i$$

is the proportion of the population that has fitness greater than w (The standard cumulative distribution function would be $1 - \overline{F}(w)$).

First several definitions need to be made.

• Let $\mathcal{R}(w)$ be the probability that random search produces an individual fitter than w.

• For a parental pair $\{j, k\}$, define the *search bias*, $\beta_{jk}(w)$, to be the excess in their chance of producing offspring fitter than w, compared with random search:

$$\beta_{jk}(w) = \sum_i F_i(w)\ T(i \leftarrow j, k) - \mathcal{R}(w).$$

• The average search bias for a population before selection is

$$\overline{\beta}_u(w) = \sum_{jk} \beta_{jk}(w)\ x_j\ x_k.$$

• The variance in the (relative) fitness of parental pairs is

$$\text{Var}[w_j\ w_k/\overline{w}^2] = \sum_{jk} \left[\frac{w_j\ w_k}{\overline{w}^2}\right]^2\ x_j\ x_k - 1 = \text{Var}[w_i/\overline{w}](2 + \text{Var}[w_i/\overline{w}])$$

• The coefficient of regression of $\beta_{jk}(w)$ on $w_j\ w_k/\overline{w}^2$ for the population, which measures the magnitude of how $\beta_{j,k}(w)$ varies with $w_j w_k/\overline{w}^2$, is

$$\text{Reg}[\beta_{jk}(w) \rightarrow w_j\ w_k/\overline{w}^2] = \text{Cov}[\beta_{jk}(w),\ w_j\ w_k/\overline{w}^2]\ /\ \text{Var}[w_j\ w_k/\overline{w}^2].$$

Theorem 2 (Evolution of the Fitness Distribution)
The probability distribution of fitnesses in the next generation is

$$\overline{F}(w)' = \mathcal{R}(w)\ +\ \overline{\beta}_u(w)\ +\ \text{Var}[w_j\ w_k/\overline{w}^2]\ \text{Reg}[\beta_{jk}(w) \rightarrow w_j\ w_k/\overline{w}^2]. \tag{3.6}$$

Theorem 2 shows that in order for the GA to perform better than random search in producing individuals fitter than w, the search bias, plus the parent-offspring regression scaled by the fitness variance,

$$\overline{\beta}_u(w) + \text{Reg}[\beta_{jk}(w) \rightarrow w_j\ w_k/\overline{w}^2]\ \text{Var}[w_j\ w_k/\overline{w}^2], \tag{3.7}$$

must be positive.

We wish to know the frequency with which the population produces individuals that are fitter than any that exist. Here one simply uses:

$$w = w_{\max} = \max_{i:x_i>0}(w_i)$$

in Theorem 2. But this can be further refined by quantifying the degree to which the transmission function is *exploring* the search space versus *exploiting* the current population. Holland [1975] introduced the idea that GAs must find a balance between exploration and exploitation for optimal search. For any transmission function, a global upper bound can be placed on its degree of exploration [Altenberg and Feldman 1987]:

Lemma 1 (Exploration Rate Limit)
Any transmission function $T(i \leftarrow j,k)$ can be parameterized uniquely as

$$T(i \leftarrow j,k) = (1-\alpha)\,(\delta_{ij}+\delta_{ik})/2 + \alpha P(i \leftarrow j,k), \tag{3.8}$$

with

$$\delta_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases},$$

where

$$\alpha = 1 - \min_{i,j\neq i}\{2\,T(i \leftarrow i,j),\, T(i \leftarrow i,i)\} \in [0,1],$$

and $P(i \leftarrow j,k)$ is a transmission function in which $P(i \leftarrow i,j) = 0$ for some i,j.

The value α is defined as the global exploration rate limit *for the transmission function, since it sets a maximum on the rate at which novel genotypes can be produced by transmission for all possible population compositions. The exploratory part of the transmission function, $P(i \leftarrow j,k)$, is referred to as the* search kernel.

In classical GAs, for example, where the transmission function consists of recombination followed by mutation, the transmission matrix $\boldsymbol{T}$ from (3.3) can be decomposed into the product

$$\boldsymbol{T} = [(1-\mu)\boldsymbol{I} + \mu\boldsymbol{M}]\,[(1-\rho)\,(\mathbf{1}^{\mathrm{T}} \otimes \boldsymbol{I}) + \rho\boldsymbol{R}],$$

where $1-\rho$ is the chance of no crossover and $1-\mu$ is the chance of no mutation during reproduction, $\boldsymbol{M}$ is an n by n Markov matrix representing mutation probabilities, $\boldsymbol{R}$ is an n by n^2 matrix representing recombinant offspring probabilities, and $\mathbf{1}$ is the n-long vector of 1s, yielding

$$\alpha = \mu + \rho - \mu\rho.$$

Using (3.8), recursion (3.1) can be rewritten as:

$$x_i' = (1-\alpha)\,\frac{w_i}{\overline{w}}\,x_i + \alpha \sum_{jk} P(i \leftarrow j,k)\,\frac{w_j w_k}{\overline{w}^2}\,x_j x_k, \tag{3.9}$$

This gives:

Theorem 3 (Evolvability)
The probability that a population generates individuals fitter than any existing is

$$\overline{F}(w_{\max})' = \alpha\left\{\mathcal{R}(w_{\max}) + \overline{\beta}_u(w_{\max}) + \mathrm{Var}[w_j\,w_k/\overline{w}^2]\;\mathrm{Reg}[\beta_{jk}(w_{\max}) \rightarrow w_j\,w_k/\overline{w}^2]\right\},$$

where now the search bias $\beta_{jk}(w_{\max})$ is defined in terms of the search kernel:

$$\beta_{jk}(w) = \sum_i F_i(w)\;P(i \leftarrow j,k) - \mathcal{R}(w).$$

Proof. This result is direct, noting that for the terms times $1-\alpha$ in (3.9), $\sum_i F_i(w_{\max})\,x_i\,w_i/\overline{w} = 0$. ■

Both the regression and the search bias terms require the transmission function to have "knowledge" about the fitness function. Under random search, both these terms would be zero. It is this implicit knowledge that is the source of power in genetic algorithms.

It would appear that maximum evolvability would be achieved by setting $\alpha = 1$. However, with $\alpha = 1$, cumulative increases in the frequency of the fittest types from selection cannot occur, and the average search bias would not be able to increase. The average search bias of the population reflects the accumulation over time of the covariance between the search bias and fitness, as seen by how $\overline{\beta}_u(w)$ itself changes in one generation:

$$\begin{aligned}\overline{\beta}_u(w)' &= \sum_{j,k} \beta_{jk}(w)\,x_j'\,x_k' \\ &= (1-\alpha)^2\left[\overline{\beta}_u(w) + \mathrm{Cov}[\beta_{jk}(w), \frac{w_j w_k}{\overline{w}^2}]\right] + O(\alpha)\end{aligned} \tag{3.10}$$

The optimal value of α is therefore a complex question, but has received a good amount of theoretical and empirical attention for special cases [Hesser and Männer 1991, Bäck 1992, Schaffer *et al.* 1989, Davis 1989, Grefenstette 1986].

3.2.4 Price's theorem and the Schema Theorem

None of these theorems on GA performance needed to invoke schemata. The Schema Theorem [Holland 1975] has been viewed as the theoretical foundation explaining the power of genetic algorithms. However, there is nothing in the Schema Theorem that can distinguish a "GA-easy" problem from a "GA-hard" problem; random and needle-in-a-haystack fitness functions will still obey the Schema Theorem, in that short schemata with above average fitness will increase in frequency, even though schemata are unrelated in any way to the fitness function. Mühlenbein [1991] and Radcliffe [1992] have also pointed out that the Schema Theorem does not explain the sources of power in genetic algorithms.

In expositions of how "schema processing" provides GAs with their power, and in the framing of the Building Blocks hypothesis [Goldberg 1989], it is implicit that the regression and search bias terms given in the theorems here will be positive under the recombination operator. In "deceptive" and other GA hard problems, at some point in the population's evolution before the optimum is found, the evolvability vanishes, i.e. the terms in (3.7) go to zero or below. Investigations on the performance of different representations and operators are implicitly studying how to keep (3.7) positive until the optimum is attained.

For example, Hamming cliffs – where single bit changes in classical binary string GAs cause large changes in the real valued parameter encoded by the string – are a problem because large jumps in the parameter space will have lower correlations in fitness for typical problems. Gray code increases the correlation resulting from single bit changes on average.

The fact that theorems 1 to 3 apply to any representation scheme and genetic operator is important, because they give the measure by which representations and operators should be judged. Under some representations, mutation may give greater evolvability over the search space; under other representations, recombination or some other operator may give greater evolvability. The operators that gives the greater evolvability may change as the population evolves.

Moreover, the question of whether non-binary representation or non-classical operators, (i.e. not recombination and mutation) are "legitimate" because they do not fit into the framework of "schema processing", should be seen as a misconception. Any combination of operators and representations that achieve the same evolvability properties over the course of the run are equally good and legitimate.

At least two studies [Manderick, de Weger, and Spiessens 1991, Menczer and Parisi 1992] have attempted to estimate the performance of different genetic operators by examining the correlation in mean fitness between parent and offspring. What Price's theorem shows is that this correlation (as embedded in the covariance expression) is an exact term in the equation for the change of the mean fitness of the population. In a model of a real-valued GA, Qi and Palmieri [Palmieri and Qi 1992, Qi and Palmieri 1992] go further and actually

derive a version of Price's theorem in order to give the change in the mean phenotype of the population, using the vector-valued phenotype as the measurement function.

3.3 Dynamics of Genetic Programming

In this section, I look into how the search bias and parent-offspring regression themselves can evolve through the differential proliferation of blocks of code within programs. The dynamics of code proliferation in genetic programming are, in their exact behavior, a formidable analytical problem. So I will describe what I believe are their qualitative aspects, and analyze a simplified model for the dynamics of GP hill-climbing.

My basic conjecture is that blocks of code proliferate within programs based on the shape of the *distribution* of their fitness effects when added to the programs in the population. The differential proliferation of code can then come to change the distribution of fitness effects resulting from the genetic operator so that the evolvability itself will evolve.

3.3.1 A model of genetic programming dynamics

Here I assume that the only genetic operator is recombination, and no mutation is acting. Let $\mathcal{B}$ be the space of different blocks of code. Let $C(i|j)$ be the probability that the recombination operator picks out a particular block i from a program j. So $\sum_{i\in\mathcal{B}} C(i|j) = 1$, for all $j \in \mathcal{S}$. Then the frequency, p_i, that the operator picks block i from a random program in the population is

$$p_i = \sum_{j\in\mathcal{S}} C(i|j)\, x_j,\ i \in \mathcal{B}.$$

The *marginal fitness*, u_i, (or if one prefers, "schema fitness") of block i is the average of the fitnesses of the programs that contain it, weighted by the frequency of the block within the programs:

$$u_i = \frac{1}{p_i} \sum_{j\in\mathcal{S}} C(i|j)\, w_j\, x_j,\ i \in \mathcal{B}.$$

With these definitions, one can give (3.9) in a GP form:

$$x'_i = (1-\alpha)\,\frac{w_i}{\overline{w}}\,x_i + \alpha \sum_{j\in\mathcal{S}} \sum_{k\in\mathcal{B}} P(i \leftarrow j,k)\,\frac{w_j}{\overline{w}}\,x_j\,\frac{u_k}{\overline{w}}\,p_k. \tag{3.11}$$

Note that $P(i \leftarrow j,k)$ is no longer symmetrical in arguments j and k.

3.3.2 The "constructional" fitness of a block of code

As described in section 3.1.4, under high recombination rates, a block of code in a program that is less fit than average, and on its way to extinction, may nonetheless get sampled and added to another program, so blocks with a high probability of producing a near neutral effect on fitness may be able to proliferate when competing against blocks with large portions of deleterious effects.

Under low recombination rates, however, the population would tend to fix on the single fittest program between recombination events. Deleterious recombinants would usually go extinct before they could be sampled for further recombination. So blocks of code could proliferate only by producing fitness increases when added to the program fixed in the population. In this much more simple situation, which is effectively "GP hill-climbing", I will show how the program can come to be composed of code that has a high chance of producing fitness increases when duplicated within the program.

To model differential code proliferation in genetic programming, I need to make additional simplifying assumptions. First, I assume that blocks of code are transmitted discretely and not nested within one another. Second, I shall consider only recombination events that add blocks of code, without deleting others. Third, there must be some sort of "regularity" assumption about the properties of a block of code in order for there to be any systematic differential proliferation of different blocks. Here I shall assume that when a given block of code is added to a program, the factor by which it changes the program's fitness has a relatively stationary distribution. This is necessary for it to have any consistent constructional fitness as the population evolves. In reality, this distribution doubtless depends on the stage of evolution of the population, the diversity of programs, and the task domain. These complications are, of course, areas for further investigation.

When a new block of code of type i is created, suppose that, with probability density $f_i(\omega)$, it multiplies the fitness of the organism by a factor ω. This probability density would be the result of the phenotypic properties of the block of code and the nature of the programs it is added to as well as the task domain.

Under small α, the population evolves only when the addition of a block of code increases the fitness of the program. Then, before other code duplications occur, the population will have already fixed on the new, fitter program. The probability, ξ_i, that a copy of a block of code, i, increases the fitness of the program is

$$\xi_k = \int_1^\infty f_k(\omega)\, d\omega = \sum_i F_i(w_j)\, P(i \leftarrow j, k) = \beta_{jk}(w_j) + \mathcal{R}(w_j), \tag{3.12}$$

for block k being added to program j to produce program i. My major simplifying assumption is that ξ_k is a constant for all programs j it is added to. The value ξ_k is the rate

that a block of code gives rise to new copies of itself that are successfully incorporated in the program, and one can therefore think of ξ_k as block k's "constructional fitness".

Let $n_i(t)$ be the number of blocks of code of type i in the program at time t, and set $\alpha \approx 0$ to be the rate of duplication per unit time under the genetic operator, for all blocks of code. The probability that the operator picks block i for duplication is

$$p_i(t) = n_i(t) \,/\, N(t), \text{ where } N(t) = \sum_i n_i(t).$$

Using the probability, ξ_i, that a copy of block i is successfully added to the program, one obtains this differential equation for the change in the composition of the program (approximating the number of blocks of code with a continuum):

$$\frac{d}{dt} n_i(t) = \alpha\, \xi_i\, n_i(t) \,/\, N(t). \tag{3.13}$$

Within the evolving program, the ratio between the frequencies of blocks of code grows exponentially with the difference between their constructional fitnesses:

$$\frac{n_i(t)}{n_j(t)} = \frac{n_i(0)}{n_j(0)}\, e^{\alpha\,(\xi_i - \xi_j)\, \int_0^t d\tau / N(\tau)}.$$

Theorem 4 (Evolution of Evolvability)
The evolvability of the program, which is the average constructional fitness of its blocks of code,

$$\overline{\xi}(t) = \sum_i \xi_i\, p_i(t),$$

— i.e., the chance that a duplicated random block of code increases the fitness of the program — increases at rate

$$\frac{\alpha}{N(t)} \mathrm{Var}(\xi) > 0.$$

Proof.

$$\begin{aligned} \frac{d}{dt}\overline{\xi}(t) &= \sum_i \xi_i \frac{d}{dt}\left(\frac{n_i(t)}{N(t)}\right) = \frac{\alpha}{N(t)^2}\sum_i \xi_i^2\, n_i(t) - \frac{\alpha}{N(t)^3}\left(\sum_i \xi_i\, n_i(t)\right)^2 \\ &= \frac{\alpha}{N(t)}\left[\sum_i \xi_i^2 p_i(t) - \overline{\xi}(t)^2\right] = \frac{\alpha}{N(t)}\mathrm{Var}(\xi). \end{aligned}$$

■

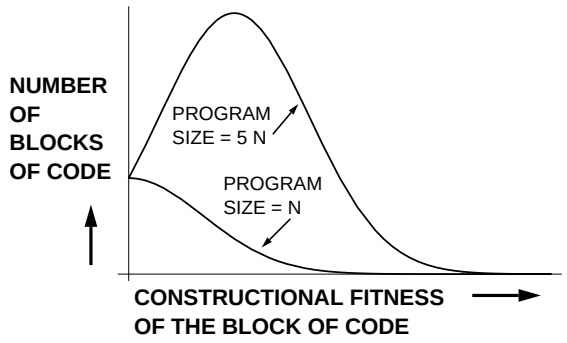

Figure 3.1
A comparison of the distribution of constructional fitnesses of blocks of code within the evolving program. Beginning with an initial program of size N at generation 0, a truncated normal distribution is assumed as an illustration. Under the dynamics of constructional selection, by the time the program has evolved to size $5N$, the distribution has shifted in favor of more constructionally fit blocks of code.

Thus, the rate of increase in the evolvability of a program is proportional to the variance in the constructional fitnesses of its blocks of code. This result is Fisher's fundamental theorem of Natural Selection [Fisher 1930], but applied not to the *fitness* of the evolving programs, but to their *evolvability*, i.e. the probability that genetic operators acting on the program gives rise to new fitter programs.

The distribution of constructional fitnesses of the blocks of code of the program can be followed as the program grows in size through evolution according to (3.13). An example is plotted in figure 3.1, where the distribution of constructional fitness values among the blocks of the program in generation 0 is compared with the distribution when the program has evolved to 5 times its original size. The blocks with high likelihood of increasing program fitness when duplicated have increased their proportion enormously.

3.3.3 The constructional fitness is distinct from the marginal fitness

This is illustrated with the following example. Consider two programs of equal fitness, each of which has only one subtree which has any chance of increasing its fitness when duplicated. Call the subtree in the first program A, and the subtree in the second program B. All the rest of the code in both programs produces only deleterious results when duplicated.

Suppose that only 0.001 of the possible ways subtree A is duplicated within the program increases its fitness, and suppose that for subtree B the chance is 0.0001 in its program. The values 0.001 and 0.0001 are the constructional fitnesses of the two subtrees. But suppose further that when duplications of subtree A produce a fitness increase, the increase is by a factor of 1.1, while for subtree B that factor is 10. One can then follow the expected

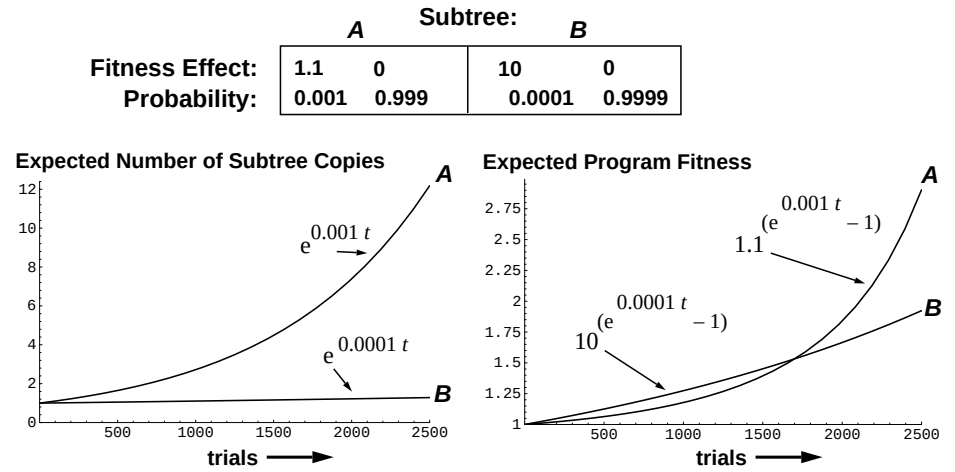

Figure 3.2
Example showing that the "constructional" fitness of a subtree is not the same as its marginal fitness in the population. The equations are given for each plot. On the left, $n_i(t) = e^{\xi_i t}$, $i \in \{A, B\}$. On the right, $w(t) = \sigma_i^{[\exp(\xi_i t)-1]}$, where σ is the fitness effect factor for the subtree.

number of copies of the two subtrees, and the expected fitnesses of the programs carrying them over time, using the model (3.13). This is plotted in figure 3.2.

As one can see, the program whose subtree, A, has the higher constructional fitness but the smaller fitness effect, falls behind in its expected fitness at first, but later shoots ahead of the program with subtree B, as exponential growth in A takes off. This assumes that the proportion of the programs that are block A or B is still small. Therefore, the constructional fitness is not the same as the marginal fitness of a block of code, and the Schema Theorem does not account for the proliferation of blocks of code within programs.

3.4 Discussion

3.4.1 Conservative versus exploratory code

In mature GP populations with typical high recombination rates, the blocks of code that proliferate may not be those that have the greatest chance of increasing fitness when added to programs, but may be those that play a more conservative game and trade-off their ability to increase fitness with an ability not to be deleterious when added, e.g nearly neutral code. Thus the constructional fitness of a block of code may not correspond to its evolvability

value. This is because high recombination rates induce selection on the central part of a block's fitness distribution as well the upper tail. One can expect that the upper tail (i.e. the probability of a block increasing fitness when added) would shrink as the population becomes more highly adapted, so that the average fitness effect would become a more important statistic for a block of code. It should be possible to measure this in GP runs.

Low recombination rates, while probably not optimal for GP performance, have the benefit that constructional selection will favor blocks of code with the fattest upper tails for their fitness effects, yielding code with greater evolvability. The following ways to limit constructional selection to the upper tails of the distribution under high recombination rates suggest themselves:

"Upward-mobility" selection. When a recombinant offspring is produced, place it in the population only if it is fitter than its parents. This could be implemented with the requirement that it be fitter that both parents, one parent, a mid-parental value, or a supra-parental value. This regime could be exaggerated by adding to the population more than one copy of individuals who exceed their parents' fitnesses by a certain amount. Upward-mobility selection would put deleterious and near-neutral recombinations on an equal footing, thus preventing the proliferation of code that trades-off its evolvability value for an ability not to produce deleterious effects.

Soft brood selection. Suppose one were to produce a number of recombinant offspring from each mating, and use only the fittest of the "brood" as the result of the mating (effectively holding a tournament within the brood). This would be called soft brood selection in population biology — "soft" because selection within the brood does not affect how many offspring the mated pair contribute to the next generation.

If the total probability in the upper tail of the fitness effect distribution for block i were ξ_i, as in (3.12), then with a brood size of n, the probability that the offspring is fitter than its parent (the recipient of the block of code) would be $1-(1-\xi_i)^n \approx n\xi_i$ for ξ_i small. Therefore, scaling $n \approx 1/\xi_i$ as the population evolves should mimic the dynamics of constructional selection under small recombination rates. Thus one has the following:

Conjecture: Upward-mobility Selection, Brood Selection, and Evolvability
Under soft brood selection or upward-mobility selection, the distribution of fitness effects from recombination events in genetic programming should increase in both the upper and lower tails compared with conventional selection.

Tests of this conjecture are currently under way. Whether brood selection would be of practical benefit depends on whether the computational overhead of producing larger broods is not better expended on larger population sizes.

"Genetic engineering". A more direct way to counter constructional selection for conservative code is create intelligent recombination operators that seek out code with high evolvability value. This can be done by estimating the evolvability value of different blocks of code based on the sampling history of that code, creating a library of the code with highest evolvability values, and having the operator draw from that library as the source of donor code. This approach is also under development by Walter Tackett (personal communication). Constructing the library requires keeping tallies of the distribution of fitness changes produced by each genetic operation in order to estimate a block of code's evolvability value. It may be profitable as well to make the genetic operator more intelligent by adding context sensitivity. It could examine the arguments and calling nodes of a block of code to see which of them it increased fitness most often with.

3.4.2 Principle 1 applied to genetic programming

In the design of genetic programming algorithms, the point of Principle 1 may be helpful: changes in representations or operators are relevant to the evolutionary dynamics only through how they change the transmission function. This point can be obscured because representations serve a dual function in GP: the representation is used as a computer program to perform the computations upon which fitness will be based, and the representation is operated upon by the genetic operators to produce variants. These two functions of the representation I refer to as *generative* and *variational* aspects, respectively, and it is the variational aspect that is relevant to the evolutionary dynamics.

Two modifications to GP representation provide an illustration: "modules" [Angeline and Pollack 1992] and "automatically defined functions" (ADFs) [Koza 1993].

Angeline and Pollack introduce an operator which replaces a subtree with a single referent (possibly taking arguments), making the subtree into an atom, which they call a "module". The same evolutionary dynamics would result from adding markers that told the genetic operator to avoid crossover within the delimited region. The compression operator therefore changes the variational aspect of the representation.

Koza [1993] has introduced ADFs as a modification to GP representation which has been shown to improve its evolvability by orders of magnitude in a number of task domains. How can we understand why ADFs work? A way to see what ADFs are doing to the variational properties of the representation is to expand the "main" result-producing branch of the program by recursively replacing all ADF calls with the code being executed. Let us then consider what the operator acting on the ADF form of the representation is doing to the expanded form of the representation.

First off, in generation 0, where one has only new, randomly generated programs, the expanded form would show internally repeated structures of code. This would be

vanishingly unlikely for random programs where each node and argument were sampled independently from the pool of functions and terminals. But this effect could be achieved by non-independent generation of the branches of the initial population.

In the ADF form of the representation, when the ADF branch undergoes recombination, in the expanded form one would see parallel substitutions in each instance of the repeated code. Thus an operator which recognized repeated instances of a block of code and changed them in parallel would have this same variational property as ADFs. One can translate between two equivalent forms — simple operators acting on more complex representations and more complex operators acting on simple representations.

The success of ADFs indicates that, for a variety of cases, the dimensions of variation produced using ADFs are highly attuned to the dimensions of variation of the task domain. A main research task is to further explore the variational possibilities of representations and operators to find those that are attuned to their task domains. A few suggestions are offered below:

Divergent ADFs. Allow different calls to an ADF to be replaced with their own instantiation of the ADF code, which can evolutionarily diverge from the original ADF. This mimics the process of gene duplication and divergence in biological evolution. The ADF that may be optimal for one calling function may not be optimal for another, so relaxing the constraint may allow better optimization. Alternatively, each ADF could be instantiated by its own code, but a gene-conversion-like operator could produce a tunable degree of information exchange among the family of related ADFs.

"Cassette" recombination[4]. Crossover occurs at both the root and branch boundaries of an internal part of a subtree, so that the set of inputs feeding into a root are unchanged, but the intervening code varies (This genetic operator grew out of a discussion with Kenneth Kinnear where we jointly defined the operator; he calls this operator "modular crossover" in Chapter 6 of this volume). Maintaining the relationship between distal arguments and proximal functions while varying the processing in between might be an advantageous dimension of variation.

Branching and junctions. Included with the set of terminals and functions could be primitives that facilitate the creation of programs with reticulated topologies. A branch would allow the value returned by a subtree to be piped as arguments to several different functions in the program.

Scaling-up. The problem of scaling-up from small "toy" problems to large ones is ubiquitous in artificial intelligence. The ability to scale-up may emerge from genetic programming if one can manage to make scaling-up an evolvability property. This could be attempted by maintaining a constant directional selection for performing at an increasing

[4]This term is borrowed from genetics, where it refers to the substitution of one block of genes by another through transposition; yeast use it in mating-type switching, for example [Egel, Willer, and Nielsen 1989].

scale: when the population reached a certain performance level for fitness cases at one scale, fitness cases at the next higher scale could be introduced. Then blocks of code may proliferate which are able to increase, through simple actions of the genetic operator, the scale which the program handles.

3.4.3 Conclusions

In this chapter I have described how an emergent selection phenomenon, the evolution of evolvability, can occur in genetic programming through differential proliferation of blocks of code within programs. In preparing the conceptual framework for this phenomenon, a mathematical analysis of evolvability was given for genetic algorithms in general. This analysis employed Price's Covariance and Selection Theorem to demonstrate how genetic algorithm performance requires that the genetic operator, acting on the representation, produce offspring whose fitness distributions are correlated with their parents' fitnesses. It was discussed how this theorem yields the necessary connection between representations, operators, and fitness that is missing from the Schema Theorem in order to tell how well the genetic algorithm will work.

With this general framework as a backdrop, I illustrated the evolution of evolvability in a simple mathematical model of code proliferation in genetic programming. Out of consideration of the dynamics of genetic programming, a number of suggestions were offered for ways to control the evolution of evolvability. These included GP hill-climbing, "upward-mobility" selection, soft brood selection, and "genetic engineering". Furthermore, other techniques were suggested to enhance the performance of genetic programming, including divergent ADFs, "cassette" recombination, branching and junctions, and scaling-up selection.

3.4.4 Further work

Certainly the various conjectures and suggestions made in this chapter provide directions for further work. But what is really needed are mathematical methods of analysis and simulation models to explore more thoroughly the population genetics of genetic programming. In this chapter I have been mainly concerned with population dynamics and not with the how the algorithmic properties of blocks of code may determine their evolvability value. This is an area that needs investigation. Some phenomena in GP will most likely be found to be primarily population-dynamic phenomena, while others will be found to be more deeply derived from the algorithmic nature of these evolving programs. Further simulations of genetic programming that implement the population dynamics, while controlling the fitness function, would be instrumental in distinguishing them.

Genetic programming, because of the open-ended complexity of what it can evolve, is potentially a rich new area for emergent evolutionary phenomena. Novel approaches will need to be developed before these phenomena, and their implications for genetic programming design, will be well understood.

Acknowledgements

The search for an alternative to the Schema Theorem began at the Santa Fe Institute's 1991 Complex Systems Summer School. Thanks to David Hall for pointing out the precedence of Price's work. This chapter benefited greatly from discussions held at the 1993 International Conference on Genetic Algorithms, and reviews of earlier versions. Thanks especially to Roger Altenberg, Kenneth Kinnear, Walter Tackett, Peter Angeline, Andrew Singleton, and John Koza. Cassette recombination was jointly invented in a discussion with Kenneth Kinnear.

Notice

Any application of the material in this chapter toward military use is expressly against the wishes of the author.

Bibliography

Altenberg, L. *A Generalization of Theory on the Evolution of Modifier Genes*. PhD thesis, Stanford University, 1984. Available from University Microfilms, Ann Arbor, MI.

Altenberg, L. 1985. Knowledge representation in the genome: new genes, exons, and pleiotropy. *Genetics* 110, supplement: s41. Abstract of paper presented at the 1985 Meeting of the Genetics Society of America.

Altenberg, L. and M. W. Feldman. 1987. Selection, generalized transmission, and the evolution of modifier genes. I. The reduction principle. *Genetics* 117: 559–572.

Angeline, P. J. 1994. Genetic programming and emergent intelligence. In K. E. Kinnear, editor, *Advances in Genetic Programming*, Cambridge, MA. MIT Press.

Angeline, P. J. and J. B. Pollack. 1994. Coevolving high-level representations. In C. G. Langton, editor, *Artificial Life III*, Menlo Park, CA. Addison-Wesley. In press.

Antonisse, J. 1989. A new interpretation of schema notation that overturns the binary coding constraint. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 86–91, San Mateo, CA. Morgan Kaufmann.

Bäck, T. 1992. Self-adaptation in genetic algorithms. In F. Varela and P. Bourgine, editors, *Toward a Practice of Autonomous Systems. Proceedings of the First European Conference on Artificial Life*, pages 263–271, Cambridge, MA. MIT Press.

Bäck, T., F. Hoffmeister, and H.-P. Schwefel. 1991. A survey of evolution strategies. In R. K. Belew and L. B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 2–9, San Mateo, CA. Morgan Kaufmann.

Barwise, J. and J. Perry. 1983. *Situations and Attitudes.* M.I.T. Press, Boston, pages 292–295.

Bergman, A. and M. W. Feldman. 1992. Recombination dynamics and the fitness landscape. *Physica D* 56(1): 57–67.

Brandon, R. N. 1988. The levels of selection a hierarchy of interactors. In H. C. Plotkin, editor, *The Role Of Behavior In Evolution*, pages 51–72. M.I.T. Press, Cambridge, Massachusetts.

Brandon, R. N. 1990. *Adaptation and Environment.* Princeton University Press, Princeton, pages 83–84.

Cavalli-Sforza, L. L. and M. W. Feldman. 1976. Evolution of continuous variation: direct approach through joint distribution of genotypes and phenotypes. *Proceedings of the National Academy of Science U.S.A.* 73: 1689–1692.

Davis, L. 1989. Adapting operator probabilities in genetic algorithms. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 61–69, San Mateo, CA. Morgan Kaufmann.

Dawkins, R. 1976. *The Selfish Gene*. Oxford University Press, Oxford.

Egel, R., M. Willer, and O. Nielsen. 1989. Unblocking of meiotic crossing-over between the silent mating type cassettes of fission yeast, conditioned by the recessive, pleiotropic mutant rik1. *Current Genetics* 15(6): 407–410.

Fisher, R. A. 1930. *The Genetical Theory of Natural Selection.* Clarendon Press, Oxford, pages 30–37.

Forrest, S. and M. Mitchell. 1993. Relative building-block fitness and the building-block hypothesis. In L. D. Whitley, editor, *Foundations of Genetic Algorithms 2*, pages 109–126. Morgan Kaufmann, San Mateo, CA.

Goldberg, D. 1989. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison Wesley.

Goldberg, D., K. Deb, and B. Korb. 1990. Messy genetic algorithms revisited: studies in mixed size and scale. *Complex Systems* 4(4): 415–444.

Grefenstette, J. J. 1986. Optimization of control parameters for genetic algorithms. *IEEE Transactions on Systems, Man and Cybernetics* SMC-16(1): 122–128.

Handley, S. G. 1994. The automatic generation of plans for a mobile robot via genetic programming with automatically defined functions. In K. E. Kinnear, editor, *Advances in Genetic Programming*, Cambridge, MA. MIT Press.

Hesser, J. and R. Männer. 1991. Towards an optimal mutation probability for genetic algorithms. In H.-P. Schwefel and R. Männer, editors, *Parallel Problem Solving from Nature*, pages 23–32, Berlin. Springer-Verlag.

Holland, J. H. 1975. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor.

Karlin, S. 1979. Models of multifactorial inheritance: I, Multivariate formulations and basic convergence results. *Theoretical Population Biology* 15: 308–355.

Kinnear, K. E. 1994. Alternatives in automatic function definition: a comparison of performance. In K. E. Kinnear, editor, *Advances in Genetic Programming*, Cambridge, MA. MIT Press.

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

Koza, J. R. 1993. Hierarchical automatic function definition in genetic programming. In L. D. Whitley, editor, *Foundations of Genetic Algorithms 2*, pages 297–318. Morgan Kaufmann, San Mateo, CA.

Levenick, J. R. 1991. Inserting introns improves genetic algorithm success rate: Taking a cue from biology. In R. K. Belew and L. B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 123–127, San Mateo, CA. Morgan Kaufmann.

Lyubich, Y. I. 1992. *Mathematical Structures in Population Genetics*. Springer-Verlag, New York, pages 291–306.

Manderick, B., M. de Weger, and P. Spiessens. 1991. The genetic algorithm and the structure of the fitness landscape. In R. K. Belew and L. B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 143–150, San Mateo, CA. Morgan Kaufmann Publishers.

Menczer, F. and D. Parisi. 1992. Evidence of hyperplanes in the genetic learning of neural networks. *Biological Cybernetics* 66(3): 283–289.

Mühlenbein, H. 1991. Evolution in time and space — the parallel genetic algorithm. In G. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 316–338. Morgan Kaufmann, San Mateo, CA.

Oakley, H. 1994. Two scientific applications of genetic programming: stack filters and non-linear equation fitting to chaotic data. In K. E. Kinnear, editor, *Advances in Genetic Programming*, Cambridge, MA. MIT Press.

O'Neil, E. and C. Shaefer. 1989. The ARGOT strategy III: the BBN Butterfly multiprocessor. In J. Martin and S. Lundstrom, editors, *Proceedings of Supercomputing, Vol.II: Science and Applications*, pages 214–227. IEEE Computer Society Press, 1989.

Palmieri, F. and X. Qi. 1992. Analyses of Darwinian optimization algorithms in continuous space. Technical Report EE-92-01, Department of Electrical and Systems Engineering U-157, University of Connecticut, Storrs, CT 06269-3157, Available by ftp from ftp@roma.eng2.uconn.edu.

Pitney, G., T. Smith, and D. Greenwood. 1990. Genetic design of processing elements for path planning networks. In *IJCNN International Joint Conference on Neural Networks*, volume 3, pages 925–932, New York.

Price, G. R. 1970. Selection and covariance. *Nature* 227: 520–521.

Price, G. R. 1972. Extension of covariance selection mathematics. *Annals of Human Genetics* 35: 485–489.

Qi, X. and F. Palmieri. 1992. General properties of genetic algorithms in the euclidean space with adaptive mutation and crossover. Technical Report EE-92-04, Department of Electrical and Systems Engineering U-157, University of Connecticut, Storrs, CT 06269-3157, Available by ftp from ftp@roma.eng2.uconn.edu.

Radcliffe, N. J. 1992. Non-linear genetic representations. In R. Männer and B. Manderick, editors, *Parallel Problem Solving from Nature, 2*, pages 259–268, Amsterdam. North-Holland.

Salmon, W. C. 1971. *Statistical Explanation and Statistical Relevance*. University of Pittsburgh Press, Pittsburgh.

Schaffer, J. and A. Morishima. 1987. An adaptive crossover distribution mechanism for genetic algorithms. In J. Grefenstette, editor, *Genetic Algorithms and their Applications: Proceedings of the Second International Conference on Genetic Algorithms*, pages 36–40, Hillsdale, NJ. Lawrence Erlbaum Associates.

Schaffer, J. D., R. A. Caruana, L. J. Eshelman, and R. Das. 1989. A study of control parameters affecting online performance of genetic algorithms for function optimization. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 51–60, San Mateo, CA. Morgan Kaufmann.

Schraudolph, N. and R. Belew. 1992. Dynamic parameter encoding for genetic algorithms. *Machine Learning* 9(1): 9–21.

Shaefer, C. G. 1987. The ARGOT strategy: adaptive representation genetic optimizer technique. In J. J. Grefenstette, editor, *Genetic Algorithms and their Applications: Proceedings of the Second International Conference on Genetic Algorithms*, pages 50–58, Hillsdale, NJ. Lawrence Erlbaum Associates.

Slatkin, M. 1970. Selection and polygenic characters. *Proceedings of the National Academy of Sciences U.S.A.* 66: 87–93.

Szarkowicz, D. 1991. A multi-stage adaptive-coding genetic algorithm for design applications. In D. Pace, editor, *Proceedings of the 1991 Summer Computer Simulation Conference*, pages 138–144, San Diego, CA.

Tackett, W. A. 1993. Genetic programming for feature discovery and image discrimination. In S. Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 303–309, San Mateo, CA. Morgan Kaufmann.

Teller, A. 1994. The evolution of mental models. In K. E. Kinnear, editor, *Advances in Genetic Programming*, Cambridge, MA. MIT Press.