Exons and Code Growth in Genetic Programming*

Terence Soule

Department of Computer Science
University of Idaho
Moscow, ID 83844-1010 USA
tsoule@cs.uidaho.edu

Abstract. Current theories regarding code growth (bloat) in genetic programming focus on the presence and growth of introns. In this paper we show for the first time that code growth can occur, albeit quite slowly, even in code that has a significant impact on fitness.

1 Introduction

The tendency of programs generated using genetic programming (GP) to grow without corresponding increases in fitness (code bloat) is well documented in the GP literature [1, 2, 3, 4, 5, 6, 7]. Growth has also been demonstrated in non-tree based evolutionary paradigms [3, 8, 9]. Current research on code growth in GP strongly suggests that it will occur in any evolutionary technique which uses variable size representations [4, 6] and Langdon has shown that growth can occur in non-population based search techniques [10]. Interestingly, Miller has found that growth did not occur with an evolutionary search using graph based structures [11, 12].

Importantly, most code growth consists of code which does not significantly contribute to a program's performance (commonly known as introns). Thus, significant resources are devoted to handling code which does not directly impact the fitness of the evolved solutions.

There are several theories regarding the causes of code growth. Most of these focus on introns, as most observed growth consists of non-contributing code. However, Luke has argued that introns are not actually the cause of code growth [7]. Additionally, Smith and Harries have shown that growth can occur in code that does influence fitness (exons) if the exons only have a negligible effect on performance [13].

We examine several different types of exons and show that growth can occur even with exons that have a significant impact on the programs' fitness.

2 Background

Code growth is a significant problem, as rapid program growth consumes considerable resources without significantly contributing to a solution. Additionally,

* This research supported by NSF EPS 80935.

the additional code may interfere with finding better solutions, since most of the code manipulation by will occur in regions of relatively low importance.

To date three likely explanations for code growth of been proposed: growth for protection [3, 4, 2], a removal bias in crossover that leads to growth[14], and a form of genetic drift towards larger solutions [15]. These three causes are not mutually exclusive and there is some evidence in support of each cause. In all three suggested causes introns are presented as playing a fundamental role.

2.1 Crossover and Program Size

In the simplest GP, with crossover and without subtree mutation, the only source of larger programs is crossover. Removing a small branch and adding a large branch creates a larger tree. Whereas removing a large branch and adding a small branch creates a smaller tree. However, if both offspring are kept the average program size does not change. Thus, the only way the average program size can increase during a GP run is if larger offspring are preferentially chosen during selection.

This is less true of subtree mutation, as depending on how the mutation is implemented it may preferentially *generate* larger (or smaller) trees. However, a 'fair' subtree mutation will generate new programs whose average size is the same as the population's average size. Again steady growth will only occur if larger offspring are preferentially chosen during selection. Notably Langdon used a version of crossover, called size fair crossover, in which the removed and added branches must be of approximately the same size [16]. In this case growth was significantly reduced, presumably because the amount of variation in the offsprings' sizes was reduced.

2.2 Types of Code

In order to understand the phenomenon of code growth it is useful to examine the types of introns produced by GP. From its inception it was realized that GP has a tendency to create programs with large sections of code that do not significantly effect the program's behavior or performance [1]. A few examples of such code include:

$$+(0 * \text{X}) \tag{1}$$

$$+(\text{X} - \text{X}) \tag{2}$$

$$+(1/\text{X}) \text{ where X} >> 1 \tag{3}$$

In each of these cases the code marked with an X does not significantly contribute to the program's behavior or performance (for case 2 it is the difference of the two sections). We are making the assumption that none of the operators have side effects. Historically the generic term 'intron' has been applied to code that doesn't have an effect and the term 'exon' to code that does have an effect. (Some authors have termed example 3 an intron, although it does have a small effect, whereas other authors consider it to be an exon.)

Although none of these sections of code have a significant effect, their other properties vary. In the first example X *can not* effect the program's behavior, regardless of how X is changed. Such code has been referred to as inviable code [14]. In the second example changing either X would effect the performance of the program. In the last example the code labeled X does effect performance, but probably to an insignificant degree. The value of X would have to be significantly changed to effect the output significantly.

Theoretical and experimental work has typically focused on the more restrictive introns. In part this has occurred because it is simpler to rigorously predict the effect (or non-effect) of code that never influences performance. Also, for some time it was assumed that the more restrictive types of code had the biggest impact on code growth. However, studies by Smith and Harries and more recently by Luke have shown that code with a very small effect on performance (example 3 above) can be as important for code growth as code with absolutely no effect [13, 7].

In this paper we test the various types of exons and demonstrate that growth can occur even in regions of code that have a significant impact on fitness.

2.3 Suggested causes of code growth

In roughly equivalent theories, Nordin and Banzhaf [3], McPhee and Miller [4], and Blicke and Thiele [2] have argued that code growth occurs to protect programs against the destructive effects of crossover. Several studies have shown that crossover is much more likely to decrease fitness than to increase fitness (destructive crossovers)[15, 17]. In addition, these studies show that a large proportion of crossover operations result in no change in fitness (neutral crossovers).

As noted above, evolved programs often contain large sections of introns that can not have a significant effect on fitness even when changed by crossover or mutation. The protective hypothesis proposes that there is an evolutionary benefit to increasing the proportion of introns, as crossover in these regions is more likely to be neutral, which is evolutionarily preferable to the destructive crossovers.

A second theory of code growth is based on the structure of program search spaces. It has been experimentally observed that for many problems the number of programs of a given fitness that are larger than a given size is much greater than the number of programs with the given fitness that are smaller than the given sizes [15]. Thus, ignoring other factors, a search is more likely to find larger programs of a given fitness than it is to find smaller programs of that fitness, simply because there are more larger programs within the search space.

The third theory of code growth is the removal bias hypothesis [14]. It assumes that removing a small subtree will have a relatively smaller effect on performance than removing a large subtree. In particular, smaller subtrees are more likely to consist of inviable code. However, if a branch is added to the middle of a section of inviable code it will, be definition, have no effect, regardless of the size of the added branch.

Thus, removing a small branch and adding a branch of any size is not likely to have an effect, whereas removing a large branch is likely to an effect, again regardless of the size of the added branch. Most changes that effect fitness are destructive. Thus, the theory hypothesizes that there is a bias in favor of offspring created by removing a small branch and against offspring created by removing a large branch. The net effect of such a bias would clearly be a general pattern of growth.

3 Experiments

Introns are important in all three of the hypothesized cause of code growth. However, as noted previously, Smith and Harries [13] and Luke [7] have shown that growth can also occur in exons that have no significant effect on fitness, such as the code X in: +(1/X) where $X \gg 1$. The question remains whether growth will occur in significant exons such as: +(1+X) where X is approximately 1. To answer this question we study growth when only significant exons are possible.

3.1 The Test Problem

For this experiment we must be able to precisely control what types of exons are possible and to eliminate all introns. This necessitates very simple functions, which in turn necessitates a very simple test problem. Our test problem is to evolve an expression with the value 10. Fitness is the absolute value of the difference between the value of the evolved expression and 10.

3.2 The Genetic Program

We use a simple, generational GP, written in C++. The programs are tree structured. The only function (internal node) used is a +, so the trees are binary. The 90/10 rule is used in selecting crossover points to be consistent with most other GP paradigms. (90% of the selected crossover points are internal nodes, 10% are leaf nodes.) Mutation consists of mutating an individual node into a random node of the same type (internal or leaf). However, mutation has no effect on internal nodes because only one function (+) is used. Leaf nodes are randomly changed into one of the other terminals. For some of the experiments only a single leaf type is used, in these cases mutation has no effect. Other details of the GP are shown in Table 1.

3.3 Function and Terminal Sets

In general, the functions and terminals determines what types of exons and introns can evolve. In our experiment the only function is + so only exons are possible. Our experiments consist of trying several different terminal sets and examining the resulting growth. The function/terminal sets used are:

Table 1. Summary of the Genetic Program parameters.

Objective	Find an expression with the value 10
Function Set	+
Terminal Set	varies
Population Size	800
Crossover Probability	0.9 (0.1 are copied without crossover)
Mutations Probability	0.001
Selection	3 member tournament
Generations	100
Maximum Tree Size	None
Elitism	2 copies of the best individual are preserved
Initial Population	Ramped, half-and-half
Number of trials	50

{+, 1.0} The only function is +; the only terminal is the constant 1.o. A perfect solution consists of 10 terminal nodes (1's) and 9 functions (+'s). There are a large number of binary trees that produce a perfect solution, but they are all the same size: 19 nodes (but not the same depth). It should be clear that introns are not possible with this function/terminal set. In addition, all code has the same 'fitness value', changing the size of a program by N nodes will always change the expression's value by N/2.

{+,0.5, 1.0} This is a slightly more complicated set. Introns are not possible, but there is an option between 1's and 0.5's. Clearly a larger (optimal) program is possible using 0.5's instead of 1's, so some code growth can occur. In particular, an optimal program of all 1's is of size 19 and an optimal program of all 0.5's is of size 39 (including +'s). Optimal programs with a mix of 0.5's and 1.0's will fall somewhere within this range.

{+,0.1,1} Again introns are impossible, but now the option is between 1 and 0.1.

{+, 0.1, 0.2, 0.4, 0.6, 0.8, 1.0} This final set gives evolution a variety of terminals to chose from, but as in the previous cases, they all produce exons.

4 Results

Figure 1 shows the average program size for each of the terminal sets. All four sets of data show similar behavior in the early generations; a rapid drop in size followed by a rapid rise. The drop seems to occur because in the initial, random population the largest programs have values furthest from 10 and are immediately removed from the population. This pushes the average size quite low. Once the largest individuals are removed the smallest individuals are the least fit and the average size rises. Both of these changes seem to be dependent on the composition of the initial population and are not related to regular code growth.

Once these initial shifts in size takes place each of the populations settles on a different size because of the difference in the average value of the terminals. As noted previously, for the set {+, 1} exactly 10 terminals (and 9 +'s) are

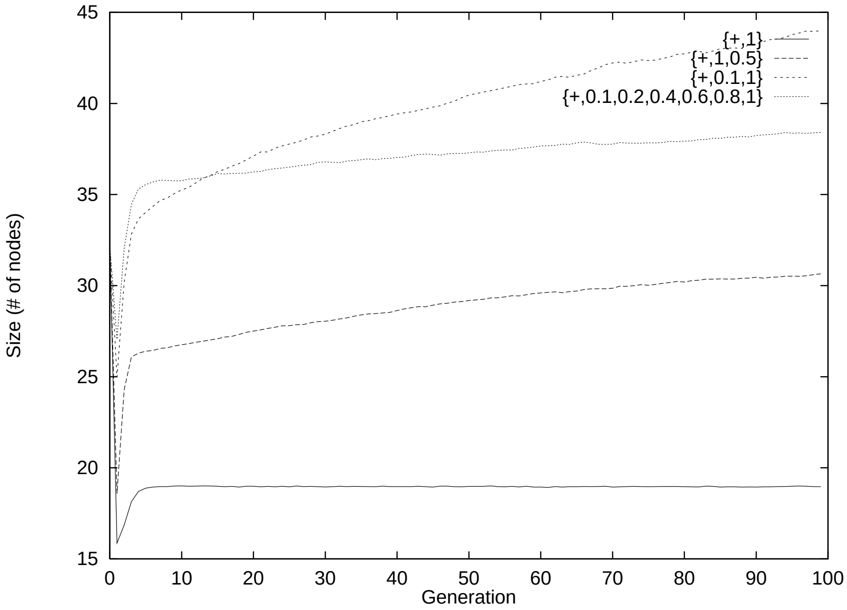

Fig. 1. Average size for each of the terminal sets. Growth occurs even when only exons are possible, but the rate depends on the exon choices.

necessary for an optimal program. Thus, as predicted, the average program size is 19. In contrast, for the set $\{+, 0.5, 1.0\}$ an average program contains an equal number of 0.5's and 1.0's. So, an average optimal program should have 12 or 14 terminals and 11 or 13 +'s. This leads to an average size of roughly 25, which is close to the results seen in Figure 1. The slight discrepancy occurs because 0.5's are favored in the first few generations. Similar reasoning applies for the other terminal sets.

Beyond these initial issues it is clear that for all of the sets (except $\{+, 1\}$) steady, slow code growth is occurring, although the rates of growth depend on the terminal set. (As noted previously with the set $\{+, 1\}$ growth is not possible without degrading fitness, so the lack of growth for that set is not surprising.) This clearly demonstrates code growth when only exons are possible.

Figure 2 shows the average fitness (difference from the target value) for each of the function sets. The values are calculated by averaging across all 800 individuals per population and across all 50 trials. Because of the very simple nature of the problem in every trail at least one optimal solution is found and preserved within the first 3 generations for every terminal set. In general, the individuals within the population converge on an optimal solution very quickly; after selection almost all of the programs are optimal. Thus, growth is not occurring as part of the search for an optimal solution. Although the population has converged on an optimal solution, Figure 2 shows a non-zero average error because the data is recorded after crossover and mutation. These operations

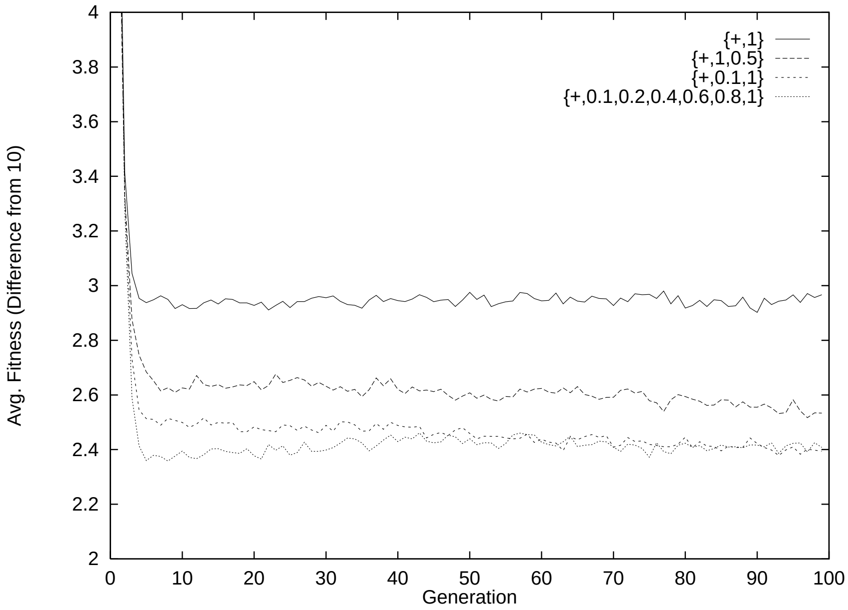

Fig. 2. Average fitness for each of the terminal sets. Fitness remains constant or improves slightly.

tend to degrade the optimal programs. Thus, the non-zero fitness corresponds to the average degradation produced by crossover and mutation in the optimal programs.

Figure 2 shows that the fitnesses are fairly constant, although the average fitnesses for the sets $\{+, 1, 0.5\}$ and $\{+, 1, 0.1\}$ appear to be improving slightly (the error is getting smaller). Thus, the data show a slow increase in size with almost no change in fitness. The only way this can occur is if the larger terminals are being selectively replaced by more of the smaller terminals. E.g. 1's are replaced by 0.5's in the set $\{+, 0.5, 1.0\}$. This observation is confirmed for the set $\{+, 0.5, 1.0\}$ in Figure 3. The percentage of 1's decreases, while the percentage of 0.5's increases. Similar results were observed for the set $\{+, 0.1, 1\}$.

Thus, we see that typical code growth (or bloat) can occur with exons that have a significant effect on fitness, albeit much more slowly than is normally observed. This is different from the work of Smith and Harries, which only showed that exons without a significant effect on fitness produce growth.

The most interesting case of terminal replacement is for the set $\{+, 0.1, 0.2, 0.4, 0.6, 0.8, 1.0\}$. Figure 4 shows the percentage of each terminal for this case. Initially the larger terminals (1 and 0.8) are heavily favored. This probably occurs because the larger terminals are necessary to reach the target value of 10. However, these larger terminals, particularly 0.8 and 1, are quickly replaced, primarily by the terminal 0.2.

Interestingly, the smallest terminal, 0.1, is still being removed, albeit slowly.

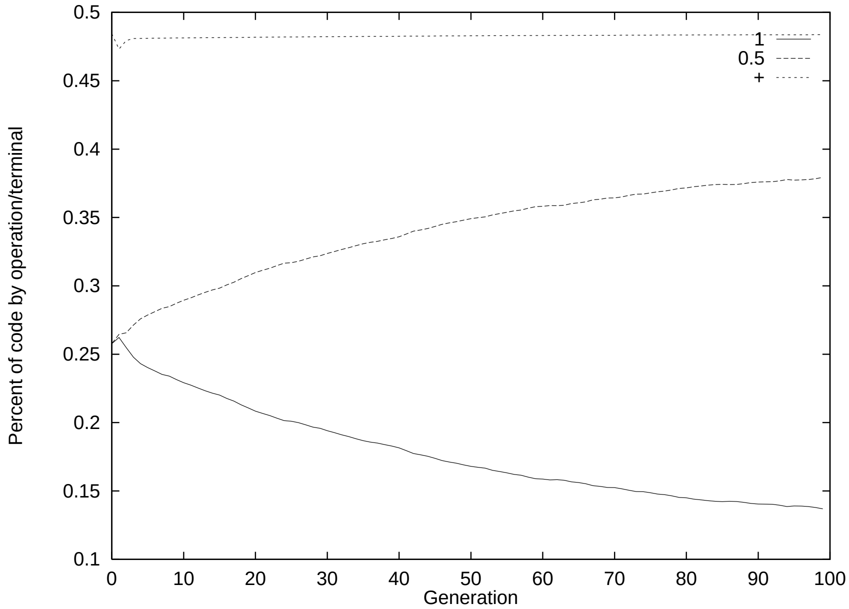

Fig. 3. Percentage of code consisting of each function and terminal for the set {+, 1, 0.5}. The 1's are being replaced by 0.5's.

One explanation is that the other terminals are all multiples of 0.2 and thus can sum to 10 in many ways without including any 0.1's; whereas a program must contain an even number of 0.1's to reach exactly 10. Thus, having 0.1's may make it less likely that a program's offspring will be optimal because of the difficulty in getting the proper number of 0.1's. This is significant because it suggests the GP is favoring programs that are likely to reach 10 *after* being mixed by crossover. Evolution is favoring programs for their *offsprings'* survivability.

5 Conclusions

First, we have clearly shown that typical code growth (or bloat) can occur with exons that significantly effect fitness, albeit quite slowly.

In our experiments, terminals with larger values are preferentially replaced by (more) terminals with smaller values. One explanation for this is that smaller terminals are less suspetiable to crossover and mutation. For this problem, changing a few 0.1's has less impact on fitness than changing a few 1's. Further evidence for this hypothesis is that growth is faster with 0.1's than with 0.5's. Following the above reasoning 0.1's would seem to be less susceptible to crossover than 0.5's, thus the evolutionary advantage of including them leads to growth. Figure 2 also supports this conclusion; as 1.0's are replaced by smaller terminals the average fitness improves implying that the programs are less damaged by crossover and mutation.

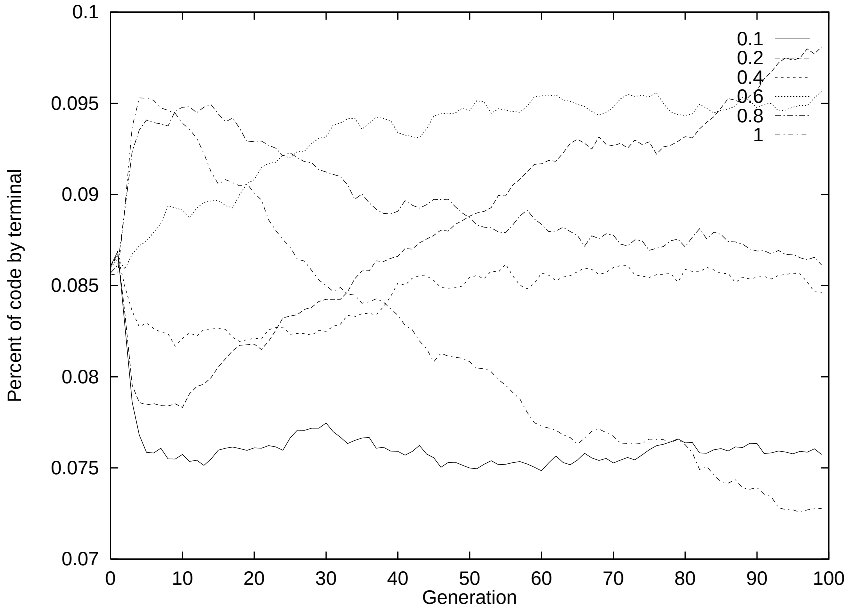

Fig. 4. Percentage of code consisting of each function for the set $\{+, 0.1, 0.2, 0.4, 0.6, 0.8, 1.0\}$. Several shifts in the percentage of each terminal are taking place.

This suggests a new term: code *stability*. More stable code is code that is less degraded by crossover and mutation. These results are preliminary evidence that in general code growth occurs to promote code stability. This idea will be addressed in more detail in future work.

We again note that for the set $\{0.1, 0.2, 0.4, 0.6, 0.8, 1.0\}$ the larger terminals are systematically replaced, but by 0.2 *not* 0.1. We hypothesized that this occurs because in a population dominated by terminals that are multiples of 0.2 having 0.1's makes it less likely that offspring will reach the value 10. Thus, we have a second example of evolution not just favoring programs with a higher fitness, but favoring programs whose offspring after crossover are likely to have a higher fitness.

References

1. John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* Cambridge, MA: The MIT Press, 1992.
2. Tobias Blickle and Lothar Thiele. Genetic programming and redundancy. In Jorn Hopf, editor, *Genetic Algorithms within the Framework of Evolutionary Computation*, pages 33 – 38. Saarbrucken, Germany: Max-Planck-Institut fur Informatik, 1994.
3. Peter Nordin and Wolfgang Banzhaf. Complexity compression and evolution. In Larry J. Eshelman, editor, *Proceedings of the Sixth International Conference on*

Genetic Algorithms, pages 310–317. San Francisco, CA: Morgan Kaufmann, 1995.
4. Nicholas Freitag McPhee and Justin Darwin Miller. Accurate replication in genetic programming. In Larry J. Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 303–309. San Francisco, CA: Morgan Kaufmann, 1995.
5. Terence Soule, James A. Foster, and John Dickinson. Code growth in genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick R. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 215–223. Cambridge, MA: MIT Press, 1996.
6. Terence Soule. *Code Growth in Genetic Programming.* PhD thesis, University of Idaho, University of Idaho, 1998.
7. Sean Luke. Code growth is not caused by introns. In *Late Breaking Papers, Proceedings of the Genetic and Evolutionary Computation Conference 2000*, pages 228–235, 2000.
8. Peter Nordin, Wolfgang Banzhaf, and Frank D. Francone. Introns in nature and in simulated structure evolution. In *Proceedings Bio-Computing and Emergent Computation.* Springer, 1997.
9. Peter Nordin. *Evolutionary Program Induction of Binary Machine Code and its Application.* Muenster: Krehl Verlag, 1997.
10. W. B. Langdon. Fitness causes bloat: Simulated annealing, hill climbing and popualtions. Technical Report CSRP-97-22, The University of Birmingham, Birmingham, UK, 1997.
11. Julian Miller. What bloat? cartesian genetic programming on boolean problems. In *Late Breaking Papers, Proceedings of the Genetic and Evolutionary Computation Conference 2001*, pages 295–302, 2001.
12. Julian Miller. Evolution of program size in cartesian genetic programming. In Lee Spector, Erik D. Goodman, Annie Wu, W. B. Langdon, Hans-Michael Voigt, Mitsuo Gen, Sandip Sen, Marco Dorgio, Shahram Pezeshk, Max H. Garzon, and Edmund Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference 2001*, page 184, 2001.
13. Peter Smith and Kim Harries. Code growth, explicitly defined introns, and alternative selection schemes. *Evolutionary Computation*, 6(4):339–360, 1998.
14. Terence Soule and James A. Foster. Removal bias: a new cause of code growth in tree based evolutionary programming. In *ICEC 98: IEEE International Conference on Evolutionary Computation 1998.* IEEE Press, 1998.
15. W. B. Langdon, Terence Soule, Riccardo Poli, and James A. Foster. The evolution of size and shape. In Lee Spector, William B. Langdon, Una-May O'Reilly, and Peter J. Angeline, editors, *Advances in Genetic Programming III*, pages 163–190. Cambridge, MA: The MIT Press, 1999.
16. W. B. Langdon. Size fair and homologous tree genetic programming crossovers. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference 1999.* Morgan Kaufmann, 1999.
17. Peter Nordin, Frank Francone, and Wolfgang Banzhaf. Explicitly defined introns and destructive crossover in genetic programming. In P. Angeline and Jr. Kenneth E. Kinnear, editors, *Advances in Genetic Programming II*, pages 111 – 134. Cambridge, MA: The MIT Press, 1996.