A Representation Scheme to Perform Program Induction in a Canonical Genetic Algorithm

Mark Wineberg, Franz Oppacher

Intelligent Systems Lab, School of Computer Science
Carleton University, Ottawa Ontario, K1S 5B6
wineberg@scs.carleton.ca, oppacher@scs.carleton.ca

Abstract. This paper studies Genetic Programming (GP) and its relation to the Genetic Algorithm (GA). GP uses a GA approach to breed successive populations of programs, represented in the chromosomes as parse trees, until a program that solves the problem emerges. However, parse trees are not naturally homologous, consequently changes had to be introduced into GP. To better understand these changes it would be instructive if a canonical GA could also be used to perform program induction. To this end an appropriate GA representation scheme is developed (called EP-I for Evolutionary Programming with Introns). EP-I has been tested on three problems and performed identically to GP, thus demonstrating that the changes introduced by GP do not have any properties beyond those of a canonical GA for program induction. EP-I is also able to simulate GP exactly thus gaining further insights into the nature of GP as a GA.

1 Introduction

In this paper we study the behavior of Genetic Algorithms on the problem of program induction. The Genetic Algorithm (GA) is an adaptive search heuristic which searches a solution space [Holland 75, Goldberg 89]. There have been many attempts to implement the task of program induction using a GA [Cramer 85, Fujiki & Dickinson 87]. The most recent attempt is Genetic Programming (GP) [Koza 92]. GP uses a modification of the GA to breed successive populations of (initially arbitrary) programs until a program that solves the problem emerges.

GP, however, does not follow the GA model exactly. It has a very different "crossover" and uses a variable structured, non-linear chromosome [Koza 92]. Consequently, the behavior of GP is not well understood, and the theoretical underpinnings of the GA model cannot be directly applied to GP [O'Reilly & Oppacher 94]. In order to understand which of the innovations utilized in GP are required to evolve a population of programs towards the correct general solution of a given task, we develop a new GA representation scheme called EP-I. EP-I encodes programs, which GP uses as variable length chromosomes, into linear chromosomes with fixed internal structures. EP-I's underlying canonical GA is then applied to the new chromosomes, and a test suite of program induction problems are solved. The success of the implementation shows that the special crossover used in GP is not necessary to solve program induction using a canonical GA[1].

---

[1] There has been a previous attempt to perform GP-like program induction with a GA [Banzhaf 93]. However in this attempt new genetic operators such as transcription, editing

2 Genetic Programming

There have been various attempts at using an explicit procedural programming language as a basis for program induction through evolution [Cramer 85, Fujiki & Dickinson 87]. The most recent and powerful of these attempts is Koza's 'Genetic Programming' [Koza 92], which is the current 'state of the art' system. GP processes parse trees using the isomorphically equivalent LISP statements (s-expressions) for its chromosomes.

At the heart of GP lie five basic insights: that it is desirable and feasible to directly manipulate parse trees that are treated as variable length chromosomes; that different problem specific primitives may have to be used for different tasks; and that the primitives should satisfy the property of closure (here closure is a design principle that any operator used in a parse tree should be able to take as input any given atomic expression (i.e., terminals), or the output of any operator). Using these insights helps alleviate the problems of both context sensitive interpretation, and order dependency[2]. The use of parse trees addresses the problem of context sensitive interpretation (at least syntactically). Since a parse tree is fully modular, any change to its structure, as long as the integrity of the tree is kept, will produce valid programs. Order dependencies are alleviated by adhering to the principle of closure in the formation of the primitives. Closure allows changes to be made in the order of the program without destroying its syntactic correctness.

Genetic Programming begins with a set of domain dependent, user-defined operators and terminals, and an initial population of randomly generated solutions. Once the initial population has been created, each program is evaluated, to see how well it operates in comparison with known output. Using the results of the fitness function, programs are selected by a standard GA selection method. Reproduction of the selected programs is done by either copying a program, or mating two programs together using a special crossover operator for parse trees (GP Crossover) . Reproduction continues until the new population is complete. This cycle of evaluation and reproduction continues until an acceptable program is created, or until a maximum number of generations have passed.

GP Crossover exchanges subtrees of two parent parse trees. The crossover points (the location of the subtrees) in the parents are chosen independently of each other. Because of this random choice of the crossover points, the heights of the subtrees are variable. Therefore the heights of the offspring may differ from those of the parents. Since it would be computationally infeasible to allow unrestricted growth in height, a maximum height restriction is imposed on the offspring of GP Crossover. If, as a result of a crossover, an offspring has a tree height that exceeds the maximum, it is rejected.

---

and repairing are introduced. This deviation from a canonical GA has to our minds the following disadvantages: it renders the existing GA theory inapplicable to the analysis of program induction, and these new unanalyzed genetic operators prevent a principled analysis of the contribution of GP's innovations.

[2] These problems are described by De Jong [De Jong 87] as inherent in any attempt to accomplish program induction using any full programming language. Order dependency occurs when the position of program structures is important. Context sensitive interpretation occurs when the meaning of an expression is affected by changes in another part of the program (this is analogous to epistasis).

3 Evolutionary Programming with Introns: EP-I

While GP is based on GA methods, there are differences. GA has fixed length chromosomes of fixed structure, and GP has non-linear, tree structured, variable length chromosomes. Accordingly GA and GP employ different operators, in particular, GA uses a mutation operator while GP does not, and the GP Crossover differs markedly from its GA counterpart.

The differences between GP, and a canonical GA, should be studied, to see which of the new, or modified, techniques, introduced by GP, are fundamental to the process of program induction through evolution, and which are merely expedient. In particular, is there anything "magical" about the use of GP Crossover that accounts for GP's demonstrated success at program induction?

There are a few other advantages of couching program induction in a canonical GA framework. Two of these advantages are: the extant GA theory becomes applicable to the program induction domain; and techniques thoroughly studied in other GA domains become available to solve problems in program induction.

The reason that GP uses a non-homologous crossover is to accommodate the non-linear structures of the parse trees. The traditional crossovers in GA, such as the 1-point crossover, are defined only for linear chromosomes. A naive application of the traditional crossovers would render the resulting offspring programs completely meaningless. In order to be compatible with the genetic operators used in a canonical GA, a chromosome must, therefore, have the following two properties:

1) the chromosome must be linear
2) after any genetic operator is applied, the resulting chromosome must produce a syntactically correct computer program.

The second property will be called *Syntactic Closure* for the rest of the paper. The proposed approach to program induction, i.e., Evolutionary Programming with Introns, *EP-I* for short, encodes the parse tree in a way that conforms to both of the above requirements.

3.1 Fixing the Tree Height

To simplify the description of EP-I, we shall for the moment restrict ourselves to the case of binary operators[3]. The property of syntactic closure requires the meaning of each locus to be fixed in the chromosome. This allows the crossover operator to exchange genes with their alleles, and not with incompatible genes. Satisfying syntactic closure presupposes the chromosome to have a fixed structure. If the structure were not fixed, the standard crossover operators would blindly cross nodes with incompatible data types. Were the chromosome to be decoded, nonsense would be produced.

As pointed out previously, GP uses a variable structured genotype. It should be noted, however, that the parse trees produced by the creation and reproduction routines have a maximum tree height. If a parse tree is produced with a greater height than the maximum it is rejected.

This suggests a way of using parse trees as the genotype and yet still keep a fixed structure for the chromosome. If one views the genome as a full binary tree,

---

[3] This restriction shall be lifted in section 3.3 below.

expanded out to its maximum depth, with the original parse tree as a subtree, then the length of the genotype is actually unchanging. Therefore the length of the chromosome is now fixed.

3.2 Full Tree Embedding of the Parse Tree

While the above observation allows one to fix the size of the chromosome, it does not determine a fixed structure within the chromosome. An natural way to embed the original parse tree in the expanded tree is at the top, node for node, leaving empty gene loci below the parse tree's terminal nodes. Unfortunately, disconnected tree fragments result if these empty nodes mutate.

A simple observation leads to a slightly different embedding of the parse tree. While the terminals of the parse tree are also its leaves, this is not true for the above embedding of the parse tree in the expanded tree. It is, therefore, intuitive that the terminals should "sink" in the expanded tree, and become leaves there too. In this embedding of the parse tree, all internal nodes are operators, all leaves are terminators, and no empty gene loci are needed. Therefore when crossover is applied, each matched pair of loci will be drawn from the same set. If the pair consists of leaves of the expanded tree, both genes are from the terminal set, otherwise, both genes are from the operator set. The internal structure of the chromosome is now static.

The embedding of the parse tree into the expanded parse tree as defined so far is obviously incomplete. The expanded tree will be mismatched with the parse tree: there will usually be more operator nodes than operators, and more leaves than terminals in the original parse tree. To solve this problem, a small set of related operators, called *selector operators*, are added to the operator set.

A selector operator is a simple function that selects one, and only one, outgoing edge for traversal. In a binary tree there are only two outgoing edges, so there are only two selector functions: selector-0 which selects the left edge, and selector-1 which selects the right edge[4]. These operators can now be used to properly embed the parse tree into the expanded tree.

The basic idea for the embedding is to use the selector operators to route the traversal of the expanded tree "around" areas that do not correspond to any part of the parse tree. Here is a more formal definition.

Let an expanded tree be a full tree with a fixed height that is greater than or equal to the height of the original parse tree. The value of each node in the parse tree must appear in the expanded parse tree. These are called the *corresponding values*. The positioning of the corresponding values in the expanded tree has the following restrictions:

1) If Node B is the k^{th} child[5] of Node A in the parse tree, then the corresponding value of node B must be in the k^{th} subtree of the corresponding value of node A.
2) Node B must be the child of Node A, or a path must exist from Node A to Node B, such that if n_i is an internal node in the path, then, if n_{i+1} is the k^{th} child of n_i and is also in the path, then the value that n_i holds is the operator selector-k. A path that has this

---

4 In case of n-ary trees, we will designate the set of selectors as 'selector-k' , k = 2,..., n.

5 This definition holds for k-ary trees where $k \geq 2$.

property, minus the two end points, will be called a *'selector chain'* (each node on the selector chain holds the selector operator that selects the next node on the selector chain).

3) The corresponding value of a terminal in the parse tree must be a leaf in the expanded tree.
4) The root of the expanded tree must hold the corresponding value of the root of the parse tree or be a selector operator that is the start of a selector chain that leads to the corresponding value of the root of the parse tree.

The nodes that are not corresponding nodes, nor in a selector chain, can hold any value, as long as the value is from the correct set: leaves must have genes from the terminal set, and internal nodes must have genes from the operator set.

If these properties hold, then the parse tree is properly embedded in the expanded tree (see figure 1 for an example of a parse tree embedded in an expanded tree). When an expanded tree has a parse tree correctly embedded in it, it is called an *expanded parse tree*.

3.3 Introns and N-ary Operators

The terms *intron* and *exon* are borrowed from Genetics [Crick 79]. For our purposes an intron shall mean any genetic material that does not have a phenotypic expression; and an exon**,** similarly, shall mean any coded genetic material that does have

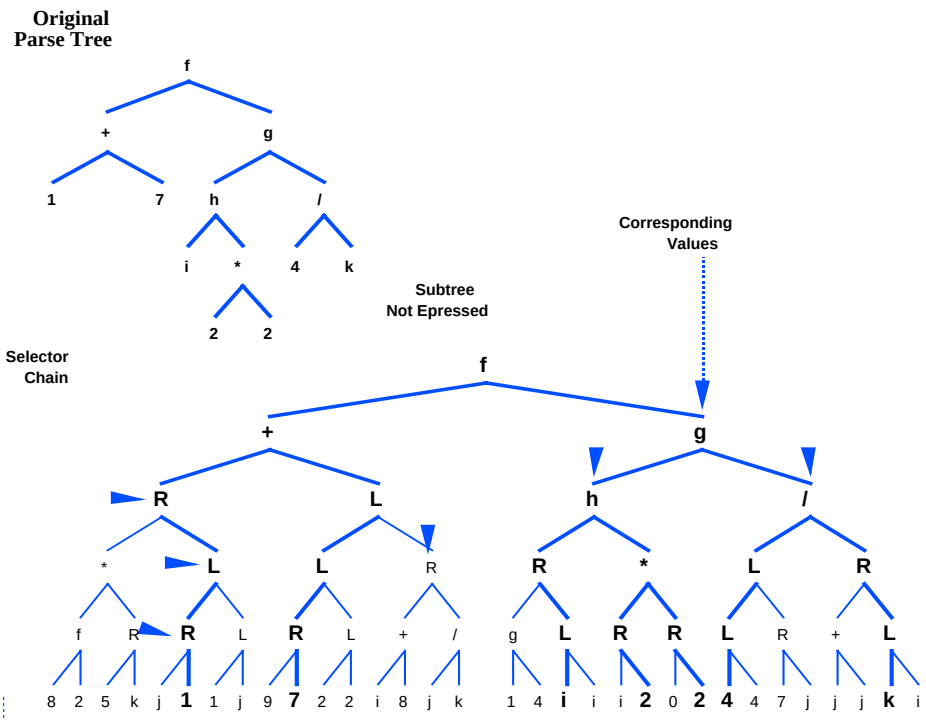


Figure 1 A Parse Tree Embedded in an Expanded Parse Tree

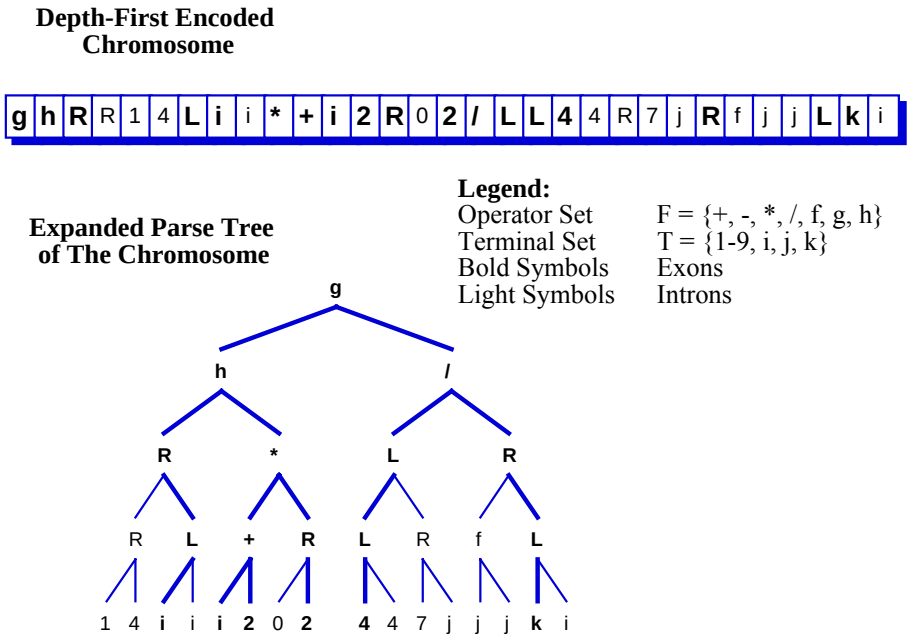


Figure 2 Example Chromosome with Introns and Exons Detailed

phenotypic behavior. The notion of introns is naturally applicable to EP-I. Like eukaryotic genomes, EP-I has genetic material inside the chromosome that is not expressed when decoded. Subtrees under edges that are not selected by a selector operator, and subtrees under a node set to an operator that does not use all of its parameters, are not included in the decoded s-expression, and so do not have an effect when that s-expression is evaluated by the fitness function. It is natural, therefore, to call these subtrees introns, and all the other nodes, which are expressed in the s-expression, exons. In other words, introns can be thought of as non-selected subtrees in the expanded parse tree. For an example of introns, and exons in a depth first encoded chromosome, see figure 2.

Introns hide genetic code from the selection process. This is both a blessing and a curse. On the negative side, this can be seen as slowing down convergence, since the selection pressures, that drive the system, are not applicable to the hidden code. Such code will not be bred out (except accidentally), even if it is useless or detrimental when expressed, nor recognized and promoted, even if it is beneficial. It is not even a good depository for safekeeping 'good' code; when hidden inside an intron, any schema can still be destroyed by the crossover operator (or the mutation operator if it is allowed to work within introns), which cannot be prevented from selecting crossover points within an intron, since the other chromosome being crossed may have an exon at that locus.

In spite of the above argument, the introns may actually help the search process. An intron can be viewed as an analog of the 'don't care' symbol explicitly expressed in the chromosome. In this respect it is comparable to the Classifier System's '#' symbol, except that introns affect more than one gene at a time. The intron dynamically blocks out segments of the genotype that are not needed for the ul-

timate solution. Since this reduces the amount of search space, the introns may actually speed up the convergence process. The positioning of the intron is influenced by selection pressure, and so introns can be seen as a large scale 'gene' that is expressed by blocking out unneeded segments of the genotype.

When an intron is turned into an exon, through mutation or crossover, the intronal material that is activated is frequently the root of a huge subtree, all of which, suddenly, is expressed / hidden. This means mutation, when introns are used in the chromosome, can have a larger effect on the resulting phenotype than in most GA systems. The effects of these large sporadic mutations are unknown.

Before the effects of introns are investigated, it is mandatory that EP-I be shown to work. The positive results reported below are encouraging, and the further exploration of the effects of introns will provide an interesting topic for future work.

Now EP-I with n-ary operators can be easily defined. The first step is to set the fan-out of each node of the tree to equal the maximum number of parameters needed by any operator in the Operator Set. Once this fan-out is found from the Operator Set, and the tree-height is known, a full tree can be created, which can be used as the template for a fixed length chromosome.

While setting the fan-out equal to the maximum number of parameters that any operator can use, fixes the structure of the expanded parse tree, there are still many operators that will take fewer parameters than the maximum. Therefore, the operators at many nodes will be given too many operands. To handle this problem, the operator can be applied to a subset of the values returned by the node's children. In other words they become introns.

3.4 EP-I Genetic Operators

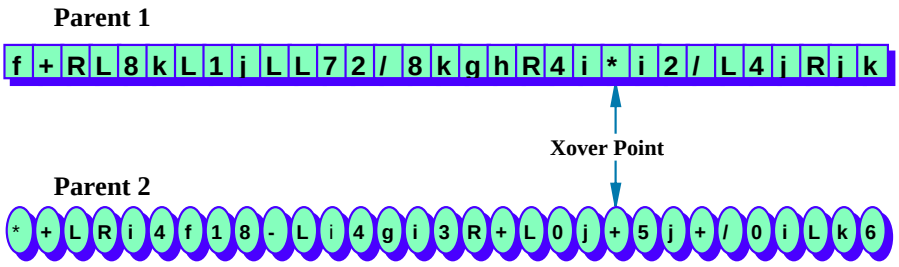
Now that the internal structure of the chromosome is fixed by EP-I, the canonical GA operator can be applied. In figure 3 we show an example of one-point crossover working on a thirty one locus chromosome which encodes an expanded parse tree of height five[6]. Just as with GP's special purpose operators, any manipulation of the chromosome with any canonical genetic operators will lead to a syntactically correct program under EP-I. This can also be seen in the figure, where the children are syntactically correct programs. Since we can now use any traditional GA operator, EP-I opens the doors for using any operators established in other contexts for the purpose of program induction.

3.5 Experiments with EP-I

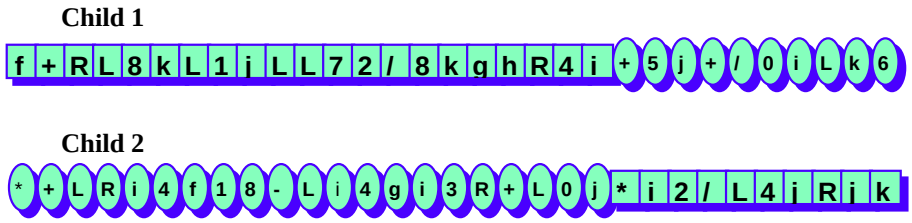
We have performed a small test suite of three of Koza's program induction problems: symbolic regression, the 6 bit multiplexor, and Fibonacci sequence induction. For a description of these problems see [Koza 92], and for a detailed account of our experiments see [Wineberg and Oppacher 94]. In general, our results using EP-I as a canonical GA are comparable to those achieved using the standard GP algorithm.

---

6 In these figures, Operator Set F = {+, -, *, /, f, g, h}, and Terminal Set T = {0-9, i, j, k}.



The decoded S-Expressions for Parent 1 and Parent 2 are:
(f (+ 1 7) (g (h i (i i 2)) (/ 4 k))) and (* (+ 4 (- i (g i 3))) (+ 0 (+ 5 j))), respectively.



The decoded S-Expressions for Child 1 and Child 2 are:
(f (+ 1 7) (g (h i (+ 5 j)) (+ (/ 0 i) k))) and (* (+ 4 (- i (g i 3))) (+ 0 (* i 2))), respectively.

Figure 3 An example of one-point crossover

Furthermore, EP-I can implement the GP crossover and therefore simulate exactly the GP algorithm. As a result of this, it becomes obvious why GP can function without the use of an explicit mutation operator. The primary use of mutation in GA is to re-introduce lost alleles at a given locus when thoses alleles have been prematurely bred out. The GP crossover, when used in EP-I, automatically moves alleles around, and so no locus can permanently be deprived of a valuable gene. Therefore the GP crossover can now easily be seen to incorporate the major effects of mutation.

4 Transposition - A Misunderstood Genetic Operator

In the course of simulating GP crossover in EP-I, we realized that the crossover was in fact moving genetic material around inside the chromosome. This is strongly analogous to a biological process called transposition.

Curiously the tranposition operator actually appears in the GA literature but is mistakenly called ‘inversion’ [Fujiki & Dickinson 87] or 'permutation' which was considered as a generalization of inversion [Koza 92]. This 'inversion' operator can be applied if the chromosome is built up of blocks of code that retain their meaning when moved to another area of the chromosome. Inversion and transposition are, in fact, very different. While both move code around inside a chromosome, transposition moves meaningful building blocks as opposed to just inverting a

sequence of single loci. Furthermore, when using transposition the general phenotype may change since the ‘transposon’ will have a slightly different effect in the new position. For example, a sorting routine, if moved to a different area of a program, will change the outcome of the program (have a different phenotypic effect) yet still can be said to 'sort its input'. The change of the phenotype after the operator has been applied is the first indication that this 'inversion' operator is, in fact, not inversion. After an application of normal inversion, the phenotype remains unchanged. True inversion is used to bring genes that work well together nearer to each other to reduce the chance of them being disassociated by one point crossover [Goldberg 89]. Transposition, however, behaves in an entirely different manner.

The transposition operator has two potential effects. The first effect, which is usually the design consideration behind its use previously, is to try out a different ordering of the transposons, hoping that a different arrangement will have beneficial phenotypic effects. The second effect is less obvious, but potentially the more potent of the two. The operator affects the distribution of transposons to other positions in the chromosome, where it might also be useful. The new placement in the current chromosome may not be very beneficial, it may even be detrimental, but from that position the transposon will be transferred to other chromosomes by crossover, alleviating the need to develop the code in those positions independently. To be effective in this way, it is imperative that the crossover will not disrupt the transposons.

It is difficult to view the first effect as having a large impact in the evolutionary process. There is no a priori reason for the system to believe that a new ordering would, statistically, have a better effect on the fitness than by inserting new genes through crossover. With crossover in place, it is just a redundant operator in the system. The second effect, however, is far more interesting. The schema being moved has a high probability of being useful, since it is from a highly fit chromosome (otherwise the chromosome it exists in would have probably not been chosen). This method alleviates the need for the same schema segment having to be re-evolved in each location where it is needed. This may be of great benefit for speeding up convergence in a GA.

To implement transposition in EP-I the following algorithm was developed. In a selected chromosome a node is randomly chosen. A second node is chosen in the same chromosome also at random but with the restriction that the new point be at the same level. The subtrees below these nodes are then exchanged, thus effecting the transposition of genetic material.

Experimentation has yet to be done to see whether this transposition method is useful in EP-I.

5 Conclusion and Future Work

To compare EP-I with GP, EP-I was run on a test suite of problems with known performance when solved by GP. Although EP-I traverses the search space of possible programs in a completely different fashion than GP, the published GP results are comparable to the results achieved by EP-I.

The success of EP-I shows that an unmodified GA can perform program induction. This also answers one of our main questions when embarking on this work: determining whether the new operators introduced by GP are uniquely responsible for its success. It is now obvious that the GP Crossover does not have any special properties

which are necessary to solve program induction, that a canonical GA lacks. Also, because of EP-I's greater representational flexibility, EP-I can explore many more properties of program induction through evolution than GP. It may even enable greater insights into the process of the GA itself since the task of program induction can now be directly compared to other searching tasks that have been implemented using GAs.

References

[Banzhaf 89] Banzhaf, W., "Genetic Programming for Pedestrians". In Proceedings of the Fiffh International Conference on Genetic Algorithms, 1993, edited by S. Forrest. San Mateo, California: Morgan Kaufmann Publishers Inc., pg. 628.

[Cramer 85] Cramer, N.L., "A Representation for the Adaptive Generation of Simple Sequential Programs". In *Proceedings of an International Conference on Genetic Algorithms and their Applications*, 1985, edited by J.J. Grefenstette. Hillsdale, NJ: Lawrence Erlbaum Associates, 183-187.

[Crick 79] Crick, F., "Split Genes and RNA Splicing" *Science*, Vol. 204 (April 20, 1979): 264-271.

[De Jong 87] De Jong, K. A., "On Using Genetic Algorithms to Search Program Spaces". In *Proceedings of the 2nd International Conference on Genetic Algorithms*, 1987, editor J.J. Grefenstette. Hillsdale, NJ: Lawrence Erlbaum Associates 210-216.

[Fujiki & Dickinson 87] Fujiki, C., and J. Dickinson, "Using the Genetic Algorithm to Generate LISP Source Code to Solve the Prisoner's Dilemma". In *Proceedings of the 2nd International Conference on Genetic Algorithms*, 1987, editor J.J. Grefenstette. Hillsdale, NJ: Lawrence Erlbaum Associates, 236-240.

[Goldberg 89] Goldberg, David E., *Genetic Algorithms in Search, Optimization & Machine Learning.* Reading, Massachusetts: Addison-Wesley Publishing Company, Inc., 1989.

[Holland 75] Holland, John H., *Adaptation in Natural and Artificial Systems*. Cambridge, Massachusetts: MIT Press, 1992; first published University of Michigan, 1975.

[Koza 92] Koza, John R., *Genetic Programming: On the Programming of Computers by means of Natural Selection* . Cambridge Massachusetts: MIT Press, 1992.

[O'Reilly & Oppacher 94] O'Reilly, U.-M., and F. Oppacher, "The Troubling Aspects of a Building Block Hypothesis for Genetic Programming". To appear in *Foundations of Genetic Algorithms 3, 1994*.

[Wineberg & Oppacher 94] Wineberg, M, and F. Oppacher, "A Canonical Genetic Algorithm Based Approach to Genetic Programming" to appear in the proceedings of the ECAI-94 Workshop on Applied Genetic and Other Evolutionary Algorithms, 1994.