Chapter 4
Self-Modifying Cartesian Genetic Programming

Simon L. Harding, Julian F. Miller and Wolfgang Banzhaf

4.1 Introduction

In some forms of genetic programming there is little distinction between genotype and phenotype. For instance, in tree-based GP [11], the genotype is a LISP expression representing a compilable LISP program, which is the phenotype. In biology, the genotype is the genetic instructions encoded in the DNA of each cell, while the phenotype is generally considered to be the physical form of the organism. This should not be confused with the behaviour of the organism. As we have seen in Chap. 2, in CGP the distinction between genotype and phenotype is much more explicit, since the genotype is decoded to a more compact form, which is the phenotype. In the decoding process, genetic instructions may be ignored, since they are not referred to in the mapping from program inputs to outputs. We have also seen that the presence of these 'junk' genes is highly beneficial to the evolutionary process (see Sect. 2.7).

Turning our attention to biology we see that in many cases (if not all), time is an essential part of the genotype–phenotype mapping. This is perhaps clearest when we consider the development of multicellular organisms. In these systems, the 'distance' between the genotype and the phenotype is enormous, as there may be a large number of cells in a phenotype. Thus, the phenotype is constructed over time, through the interaction of the genotype and the environment. Including the notion of time in the genotype–phenotype mapping may be beneficial [1].

Self-modifying Cartesian genetic programming (SMCGP) has included time by allowing new kinds of functions into the genotype. These are self-modification (SM) functions, i.e. instructions that cause alteration of the code itself. When these SM functions are obeyed, the genotype changes into a new form (i.e. develops into a new phenotype). The new phenotype can also contain SM functions, so that when these are obeyed, a further phenotype can be created. As we will see, we refer to each of these phases of obeying SM instructions as an 'iteration'.

J.F. Miller (ed.), *Cartesian Genetic Programming*, Natural Computing Series,
DOI 10.1007/978-3-642-17310-3_4,

Another aspect of biological cells is that they are responsive to an arbitrary number of inputs. So, in addition to time, we have also introduced into CGP *functions* that acquire program inputs and outputs. When used in concert with SM functions, these allow SMCGP programs to acquire more inputs and deliver more outputs. These two new features allow the possibility of CGP being able to find general solutions to problems. For instance, in CGP one can attempt to solve a particular parity problem, that is, one with a given number of inputs, whereas SMCGP allows the possibility of evolving a program which – when iterated – produces an infinite sequence of parity functions, each one having one more input than in the previous iteration. Further to this, it has already been shown that SMCGP genotypes can *provably* produce general solutions to a number of problems (e.g. parity, binary addition, and computing pi and e) [6, 7].

4.1.1 Discovering Mathematical Results Using Genetic Programming

Evolving provable mathematical results is a rarity in evolutionary computation. Streeter and Becker used tree-based GP to discover mathematical approximations to well-known mathematical series such as the harmonic series and also new Padé approximations to mathematical functions [20]. Although these were interesting formulae, they did not find exact analytical results that could be shown to converge in the limit, but they noted that finding such results would be a 'striking and exciting application of genetic programming'.

Schmidt and Lipson used GP to discover the known Hamiltonians and Lagrangians of mechanical systems purely by using GP symbolic regression techniques on data acquired through motion tracking [16]. Schmidt and Lipson also investigated using GP to solve iterated function problems (i.e. $f(f(x)) = g(x)$) and showed that one evolved function provably made $f(f(x))$ converge to $x^2 - 2$ in the limit [17].

Spector et al. showed that GP could be used to evolve hitherto unknown algebraic expressions that are important in the mathematics of finite algebras and are orders of magnitude shorter than those that could be produced by prior mathematical methods [18].

SMCGP has been able to find exact analytical results. In [6, 7], it has been shown that mathematical functions have been evolved that converge rapidly to pi and e in the limit of large iterations.

4.2 Overview of Self-Modification

Procedure 4.1 gives a high-level overview of the process of mapping a genotype to a phenotype in SMCGP. The first stage of the mapping is the modification of the geno-

type. This happens through the use of evolutionary operators acting on the genotype. The developmental steps in the mapping are outlined in lines 3–8 of the algorithm. The first step is to make an exact copy of the genotype and call it the phenotype at iteration 0. After this, the self-modification operators are applied to produce the phenotype at the next iteration. Development stops when either a predefined iteration limit is achieved or it turns out that the phenotype has no self-modification operations that are active.

At each increment, the phenotype is evaluated and its fitness calculated. The underlying assumption here is that one is trying to solve a series of computational problems, rather than a single instance as is usual in GP. For instance, this might be a series of parity functions, ever-closer approximations to pi, or the natural numbers of the Fibonacci sequence. If the problem, however, has only a single instance (i.e. a classification problem), we can take a fixed number of iterations (either a user-defined parameter or evolved) and evaluate the single phenotype. Another possibility would be to iterate until no self-modification rules are active.[1]

It is important to note that there are various ways in which there may be no active self-modification operations. Firstly, no self-modification operations may exist in the phenotype. Secondly, self-modification operations may be present but be non-coding. Thirdly, the self-modification operations may not be 'activated' when the instructions encoded in the phenotype are executed. These various conditions will be discussed in the detailed description in the following sections.

Procedure 4.1 Overview of genotype, phenotype and development

1: Generate genotype
2: Copy genotype to phenotype. Iteration, $i = 0$
3: **repeat**
4: Apply self-modification operations to phenotype i
5: increment i
6: Calculate fitness increment, f_i
7: **until** ((i equals number of iterations required) **OR** (No self-modification functions to do))
8: Evaluate phenotype fitness F from fitness increments, f_i

4.3 SMCGP and Its Relation to CGP

The genetic representation in SMCGP has much in common with the representation used in CGP. The genotype encodes a graph (usually acyclic) that includes a list of node functions used and their connections. The arity of all functions is chosen to be equal to that of the largest-arity function in the function set. So, as in CGP, functions of lower arity ignore extraneous inputs. However, there are important differences.

[1] This has not been investigated, but is likely to be problematic as the number of iterations could be very large.

Firstly, SMCGP genotypes represent a linear string of nodes. That is to say, only one row of nodes is used (in contrast to CGP, which can have a rectangular grid of nodes). Other important differences are discussed in the following sections. The evolutionary algorithm that we have used with SMCGP was the $1+4$ evolutionary strategy which is detailed in Sect. 2.6.3.

4.3.1 Self-Modification Operators

The most significant difference is the addition of self-modification (SM) operators to the function set. These operators can be used in the genotype in the same manner as the more conventional computational operators, but at run time they provide different functionality. When the SMCGP phenotype is run, the nodes in the graph are parsed in a similar way to CGP. The graph is executed recursively by following nodes from the output nodes to the terminals (inputs). When computational functions are called, then – as usual – they operate on the data coming into the node.

When an SM node is called, the process changes slightly. If an SM node is 'activated'; then its self-modification instructions are added to a list of pending manipulations which is called the *To-Do* list. The modifications in this list are then performed between iterations. Note that SM active instructions are added to the *To-Do* list in a left-to-right traversal of the encoded graph. It was decided that SM nodes should be activated in a way that is dependent on the data presented to them. If the non-SM functions were all numerical functions in nature, SM nodes are activated when the first input is greater than the second.

When solely Boolean functions are used, SM nodes are always considered to be activated.[2] The size of the *To-Do* list is chosen by the user. This importance of this is discussed further in Sect. 4.3.7.

Many SM operators are imaginable, and Table 4.1 lists the currently available operators. In the table, we can see that the operators also require arguments. These come from the genotype and are described in Sect. 4.3.3. It is also worth noting that the indices for SM operations are defined relative to the current node. This relative addressing is a fundamental part of SMCGP and is discussed in Sect. 4.3.4.

Clearly, Table 4.1 lists many functions! One of the important unresolved issues in SMCGP is deciding on a minimal SM function set.

4.3.2 Computational Functions

The computational functions used in SMCGP are typical of such functions in GP in general; however, some functions are particular to SMCGP. A nominal list of func-

[2] This is an implementation convenience that stems from the parallel implementation, where multiple bits are packed into a single word. This makes it less obvious what an analogous activation operation to that used with numerical functions should be.

Table 4.1 Definition of the self-modification functions. P_i are the evolved arguments of the self-modification functions; x represents the absolute position of the node in the graph, where the leftmost node has position 0; and c_{ij} is the jth connection gene on the node at position i

Basic	
Delete (DEL)	Delete the nodes between (P_0+x) and (P_0+x+P_1).
Add (ADD)	Add P_1 new random nodes after (P_0+x).
Move (MOV)	Move the nodes between (P_0+x) and (P_0+x+P_1) and insert after (P_0+x+P_2).
Duplication	
Overwrite (OVR)	Copy the nodes between (P_0+x) and (P_0+x+P_1) to position (P_0+x+P_2), replacing existing nodes in the target position.
Duplication (DUP)	Copy the nodes between (P_0+x) and (P_0+x+P_1) and insert after (P_0+x+P_2).
Duplicate preserving connections (DU2)	Copy the nodes between (P_0) and (P_0+P_1) and insert after (P_0+P_2).
Duplicate preserving connections (DU3)	Copy the nodes between (P_0+x) and (P_0+x+P_1) and insert after (P_0+x+P_2). When copying, this function modifies the c_{ij} of the copied nodes so that they continue to point to the original nodes.
Duplicate and scale addresses (DU4)	Starting from position (P_0+x) copy (P_1) nodes and insert after the node at position (P_0+x+P_1). During the copy, the c_{ij} of copied nodes are multiplied by P_2.
Copy to stop (COPYTOSTOP)	Copy from x to the next COPYTOSTOP or STOP function node, or the end of the graph. Nodes are inserted at the position the operator stops at.
Stop marker (STOP)	Marks the end of a COPYTOSTOP section.
Connection modification	
Shift connections (SHIFTCONNECTION)	Starting at node index (P_0+x), add P_2 to the values of the c_{ij} of next P_1 nodes.
Shift connections 2 (MULTCONNECTION)	Starting at node index (P_0+x), multiply the c_{ij} of the next P_1 nodes by P_2.
Change connection (CHC)	Change the $(P_1 \bmod 3)$th connection of node P_0 to P_2.
Function modification	
Change function (CHF)	Change the function of node P_0 to the function associated with P_1.
Change argument (CHP)	Change the $(P_1 \bmod 3)$th argument of node P_0 to P_2.
Miscellaneous	
Flush (FLR)	Clears the contents of the *To-Do* list

tions is provided in Table 4.2. A node CONST returns the first argument supplied to it. Using CONST in the function set obviates the need to have terminals providing constants (as is typical in GP). INDX outputs the position of the node in the SMCGP graph where nodes are numbered sequentially from the left, with the leftmost node being 0. INCOUNT returns the number of inputs are available to the program.

Table 4.2 Nominal computational functions. This assumes that all nodes are supplied with two inputs, a and b. Functions are protected from invalid input values where necessary

Function	Operation
NOP	No operation
DADD	$a+b$
DSUB	$a-b$
DMULT	$a \times b$
DDIV	a/b
CONST	constant (defined by P_0)
AVG	$(a+b)/2$
DSQRT	$\sqrt{a}$
DRCP	$1/\sqrt{a}$
DABS	$\lvert a \rvert$
TANH	$\tanh(a)$
TANH2	$\tanh(a+b)$
FACT	$!a$
POW	a^b
COS	$\cos(a)$
SIN	$\sin(a)$
MIN	$\min(a,b)$
MAX	$\max(a,b)$
IFLTE	if $(a < 0)$ return b, else 0
INDX	current node index
INCOUNT	number of inputs

4.3.3 Arguments

Each node in the SMCGP genotype contains three floating-point numbers. These numbers are evolved and are used in several ways by the SMCGP phenotype. The SM functions require several arguments to specify how graph modifications are to be carried out (see Table 4.1). These arguments are simply numbers, and their values are taken from the node's arguments. As the SM function arguments have to be integers, the values are truncated. However, as we saw, the arguments are also used by the CONST function. This allows SMCGP to evolve programs that contain numerical constants thus giving the program access to an arbitrary number of numerical constants. The arguments can take any value; however, when they are interpreted as parameters for self-modifying operations, values that are too large will be truncated to within the range of allowed offsets.

During iteration of the genotype, the arguments can be altered by the SM function CHP ('change parameter'). This, in principle, allows storing of the state (i.e. a memory), since a phenotype could pass information to the phenotype at the next iteration through a collection of constant values.

4.3.4 Relative Addressing

The SM operators' ability to move, delete and duplicate sections of the graph means that the classical CGP approach of labelling nodes becomes cumbersome. Classical CGP uses *absolute addressing*, so that each node has an address and nodes reference each other using these addresses (this is what connection genes are – see Sect. 2.2). In SMCGP, the structure of the phenotype program is dynamic, and to use this approach would require significant overhead in relabelling the graph at each iteration.

To simplify the representation, we therefore replace the fixed addresses with relative addresses. Now, instead of a node containing an absolute address of another node, it specifies how many nodes back from its position are required to make a connection. The connections in the genotype are now defined as positive integers that are greater than 0 (which prevents cycles). It is worth noting that classical CGP could easily have chosen this method of addressing and indeed it might have advantages in embedded CGP, since it could avoid the need to readdress the CGP fragments inside modules (see Sect. 3.2.2.1).

When the graph is run, the interpreter can calculate where a node gets its input values from by just subtracting the connection value from the current address of the node. If the node addresses a value that is not in the graph (i.e. connects too far back), then a default value is returned (in the case of numeric applications this is 0).

The arguments of SM operators are also defined relative to the current node (see Table 4.1). The relative addressing allows subgraphs to be placed or duplicated in the graph whilst retaining their semantic validity. This means that subgraphs could represent the same subfunction, but act on different inputs. This can be done without recalculating any node addresses, thus maintaining validity of the whole graph. So subgraphs can be used as functions in the sense of the ADFs of standard GP.

4.3.5 Input and Output Nodes

Most, if not all, genetic programming implementations have a fixed number of inputs. This certainly makes sense when there is a constant or bounded number of inputs over the lifetime of a program. However, it does prevent the program from scaling to larger problem sizes by increasing the number of inputs it uses – and this in turn may prevent general solutions from being found.

Similarly, most GP has a fixed number of outputs. In tree-based GP, there is typically a single output. In classical CGP, a number of input nodes are placed at one end of the graph, and these are used as the starting point for the recursive interpretation of the program.

To allow an arbitrary number of inputs and outputs, SMCGP introduces several new functions into the basic function set. These are shown in Table 4.3.

The interpreter now keeps track of an input pointer, which points to a particular input in the array of input values. Calling the function INP returns the value that

Table 4.3 Input and output functions. P_0 is the first argument gene

Function	Operation
INP	Return input pointed to by current_input, increment current_input
INPP	Return input pointed to by current_input, decrement current_input
SKIPINP	Return input pointed to by current_input, current_input = current_input + P_0
OUTPUT	Return data provided

the pointer is currently on, and then moves the pointer to the next value. When the pointer runs out of inputs, it resets to the first input. Similarly, the INPP function returns an input but then moves the pointer to the previous value. Sometimes it may not be convenient or useful to move by only one input at a time. The SKIPINP function therefore moves the pointer a number of places (specified by truncating the first floating-point argument in the node), and then returns that input.

By duplicating INP, INPP and SKIP nodes, the SMCGP phenotypes can acquire more inputs when they are iterated.

Variable numbers of outputs are handled using a similar strategy. A function OUTPUT allows the interpreter to find which nodes to use as output nodes at run time. Again, the location and number of OUTPUT nodes can change over the run time of a program. When the graph is run, the interpreter starts at the beginning of the graph and iterates over the nodes until it finds the appropriate number of OUTPUT nodes. It then evaluates (recursively) from these nodes.

For the outputs, the interpreter has features that allow it to cope when the number of OUTPUT nodes is different from the required number of outputs. If there are more OUTPUT nodes found than are needed, the excess nodes are simply ignored. If there are too few (or none) the interpreter starts using nodes from the end of the graph as outputs. This ensures that programs (of sufficient size) are always 'viable'.

4.3.6 A Simple Example

An example genotype is given in Fig. 4.1. The figure also shows, purely schematically, some phenotypes arising at different iterations.

4.3.7 Discussion

The length of the *To-Do* list is an important parameter in SMCGP. If it is chosen to be a large value, it can allow phenotypes to grow rapidly in subsequent iterations. In most experiments conducted so far (see [6]), the length of the list has been kept

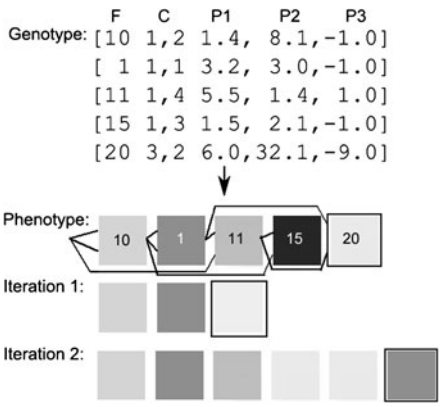

Fig. 4.1 The genotype maps directly to the initial graph of the phenotype. The genes control the number of nodes and the type and connectivity of each of the nodes. The phenotype graph is then iterated to perform computation and produce subsequent graphs. Function genes are denoted by F, connection genes by C and argument genes by P_i. The nodes in the phenotype that act as outputs are outlined.

no larger than two. At present, it is not known how the choice of this parameter affects the efficiency of the evolutionary search. When the length of the *To-Do* list is greater than one, the programs encoded in the phenotype become very difficult to understand, and so far it has only been possible to supply formal proofs of the generality of solutions when the length is one.

Another issue worthy of discussion is deciding when to apply SM operations. At present, they are added to a *To-Do* list and the SM instructions are carried out one by one until the end of the list is reached, thus producing a new phenotype. Indeed, this is what we mean by an iteration of the phenotype. However, another approach could be to carry out SM operations singly, producing a new phenotype each time, which could be evaluated in some way. This would be a more continuous form of operation and it could, at least in principle, allow more opportunities to assess the fitness of the genotype. This is effectively what happens when the length of the *To-Do* list is one.

At present, SM functions are mixed in the genotype with computational functions; thus they must act on numerical inputs. At present, we have given them passive computational roles (i.e. pass the second input). In principle, one could envisage a separation of the two aspects of computation and self-modification and introduce two separate chromosomes: a computational chromosome (CC) and a self-modification chromosome (SC). The SC would act on the CC to produce a new computational phenotype. This would be one iteration. Then, at the next iteration, the SC would be applied to the phenotype, and so on. By allowing the SM nodes in the SC to be assumed to have a single nominal input, one could use CGP to determine a graph (the SM phenotype) that would determine which SM operations

would be applied to the CC (i.e. the SC would have non-coding nodes). This could potentially make the genotype–phenotype mapping easier to reason about.

The issue of SM activation, as it stands, is problematic. In non-Boolean problems we allow the data to decide whether an SM node is active or not. Actually, in most problems thus far studied, this has not caused an issue, as only a single input has been applied at each iteration. However, we can envisage a problem which is a generalization of a symbolic regression problem, in which each instance of the problem has a table of inputs and outputs (possibly representing an unknown symbolic function). To illustrate this, consider the problem of determining the following sequence of polynomials: $f_1(x) = x$, $f_2(x) = x + 1$, $f_3(x) = x^2 + x + 1$, $f_4(x) = x^3 + x^2 + x + 1$,.... For each expression f_i, we could generate a table of values by assuming that x increments from -1 to 1, in intervals of 0.01 (say). Thus the problem to solve would be to determine the sequence of polynomials that give zero error in each instance. If SM functions were mixed with computational functions (as at present in SMCGP) and a data-dependent SM activation method was used, then different phenotypes could emerge with each data instance! Of course, evolution can find a way around this scenario when needed, perhaps by making both inputs to a SM node point to the same location. In some circumstances, for example as described in [3], the only way that the SMCGP could possibly find a learning-algorithm is to have self-modification that depends on the data. In SMCGP, if the phenotype needs to have different developmental outcomes based on the values of the computation, this is possible. It is also possible for SMCGP to ignore this feature. Further, for some problems, it can just be disabled (such as with the Boolean functions). Therefore the possibility of having data dependence seems desirable, as it provides extra functionality when it is needed, but otherwise can be ignored.

4.3.8 And Back to CGP

It is interesting to note that some of the changes to the representation are also applicable to classical CGP, and some to GP in general. The input/output strategy, relative addressing and evolved constants are all compatible with classical CGP. It remains to be seen how useful they are. However, since CGP has a fixed-length genotype even if INPUT and OUTPUT functions are included, the numbers of inputs and outputs will be bounded.

4.4 Solving Computational Problems with SMCGP: Parity

In this section we illustrate SMCGP using an example problem. The problem is how to evolve a program (or circuit) that computes even-n-parity. The parity circuit is a Boolean circuit that takes a bit string and returns TRUE if there are an even number

of 1s in that string, and FALSE otherwise (note that the case of no bits is treated as an even number).

The even-n-parity problem has been a benchmark in GP ever since John Koza discussed the problem in his first book [11]. The problem is to evolve an even-n-parity function using the Boolean two-input functions AND, NAND, OR, and NOR. Koza tackled up to 11-parity [12] using a GP system with ADFs, and found them difficult to evolve. Without ADFs, his approach failed to evolve parity circuits beyond five inputs [11]. The largest evolved parity circuit we found in the literature was 22 bits [15]. It should be noted that Poli and Page used all 16 two-input Boolean functions in their function set, whereas in the work described in this section, we used just AND, NAND, OR and NOR.

Spector also examined the even-parity problem, however he used a different function set [19]. He found better scaling behaviour than Koza did, on even-parity functions up to six inputs. Other approaches have looked at finding general solutions. Huelsbergen evolved machine-language programs that could iterate over the bits in a string and from this, parity could be easily determined [10]. The solutions would be suitable for any length of bit string. Recursion has also been successfully used to solve the parity problem [24, 23, 25]. These approaches produced programs rather than circuits to solve the problem. They also used high-level programming constructs rather than purely Boolean logical primitives.

However, we will look at a slightly different problem, namely that of evolving all even-n parity functions up to even-N parity (where N may be infinite, in which case we have a general solution). We call this problem the 'all-even-n-parity problem". In [2, 5], we investigated using SMCGP to evolve large even-parity circuits. We found that some of the evolved programs were able to generate arbitrary-size parity circuits. In [6], we gave a formal proof of the generality of one of the evolved solutions. This is reproduced in Sect. 4.4.3.

4.4.1 Definition of Fitness

With SMCGP, instead of evolving a circuit for a particular input size, we can use the technique to find a program that will generate parity functions of any size. To do this, we evolve a program that produces a two-input parity circuit on the first iteration, produces a three-input parity circuit on the second, and so on up to N bits. The fitness is the number of correctly predicted output bits. We stop testing an individual if it fails to produce a correct solution for a given input size. Procedure 4.2 details the fitness function.

We accumulated fitness over 19 iterations ($LIMIT = 19$), starting with even-2-parity and ending on even-20-parity. Subsequently we examined whether the best solutions were correct up to 24 inputs to check for signs of generalization.

Procedure 4.2 Fitness function

```
1: Fitness, F = 0
2: Copy genotype to phenotype. Iteration, i = 0
3: repeat
4:    BREAK = FALSE
5:    Apply self-modification operations to phenotype i
6:    increment i
7:    Calculate fitness on test case, f_i, by counting number of incorrect bits
8:    if f_i ≠ 0 then
9:       BREAK = TRUE
10:   end if
11:   F = F + f_i
12: until i = LIMIT OR BREAK
```

4.4.2 Results

Table 4.4 shows the average number of evaluations required to evolve a program for a given number of inputs. The success rate was 100%. The results are based on 50 trials per function set.

In Table 4.5, the results are compared with results obtained from previous CGP representations and Koza's figures for GP with ADFs [12]. The SMCGP results are clearly highly competitive. It should be remembered that SMCGP is solving the *all-even-n-parity problem* rather than a series of single instances. We have included Koza's figures for reference. Koza calculated the computational effort for a 99% success rate and so represented the number of evaluations assuming the most favourable number of runs and number of generations. More detailed comparisons between CGP and other methods have been published previously in [22]. There, it was seen that embedded Cartesian genetic programming was highly competitive with other GP methods. Poli and Page evolved solutions to even-parity for up to 22 inputs [15]; however, they only gave numbers of evaluations for a single evolutionary run when the number of inputs was 13, 15, 17, 20 and 22. Thus we could not make a comparsion.

After evolution, solutions were tested for inputs of up to 24 bits. It was found that all solutions generalized to problems of this size. Most solutions had generalized to all inputs when even-8 parity was reached. This is reflected in the results in Table 4.4, where the number of evaluations required to solve a problem stops increasing because once a solution is found to generalize, no further evolution is required.

Examining the comparative results in Table 4.5, we can see that SMCGP scales much better than CGP, ECGP and GP.

Table 4.4 Average number of evaluations required to find a program that will solve parity up to a given number of bits (50 runs)

Input bits	Average evaluations
3	247,753
4	275,663
5	278,635
6	298,104
7	318,376
8	322,843
9	322,843
10	322,843
11	322,851
12	322,851
13	322,866
14	322,866
15	322,866
16	322,866
17	322,870
18	322,870
19	322,874
20	322,874

Table 4.5 Comparison with previous work on evolving parity. 'GP' means Koza's tree-GP (with ADFs) [12], 'ECGP' means embedded CGP [21], and 'CGP' is conventional CGP. With the exception of the figures from Koza, the figures show the average number of evaluations required to find a given-sized parity circuit. Results for higher numbers of inputs are not available for CGP or ECGP. The figures from Koza represent computational effort so they represent the minimum number of evaluations required to achieve 99% success. The minimum was selected from the 'ideal' number of runs and number of generations. A dash indicates that figures are unavailable

Input bits	SMCGP	CGP	ECGP	GP
4	275,663	81,728	65,296	176,000
5	278,635	293,572	181,920	464,000
6	298,104	972,420	287,764	1,344,000
7	318,376	3,499,532	311,940	1,440,000
8	322,843	10,949,256	540,224	–

4.4.3 A General Solution to Computing Even-Parity

It is instructive to examine an evolved solution to even-n-parity and how it can be shown that the evolved genotype represents a general solution.

The genotype in Fig. 4.2 was evolved with a *To-Do* list length of 1. The 20-node genotype had only seven active nodes. The inactive nodes are shown as unconnected smaller squares. The nodes INPP at positions 0 and 2 obtain inputs x_1 and x_0, respectively. Three Boolean functions BNOR, BAND and BOR appear at positions 4, 5 and 6, respectively. The OUTPUT function obtains the single output from the BOR node. The only active SM node is DUP at position 1. It carries arguments which cause it to copy eight nodes, beginning at the node on its left (INPP), and

insert them immediately after itself. The action of the DUP node is shown using a curved line with an arrow emanating from the box. Since the genotype has no connections that are to the left of the first node, when DUP copies it disconnects the first two nodes in the generated phenotype. These appear at the beginning (left) of the new phenotype (iteration 1) at positions 0 and 1. It is important to note that in this phenotype, a previously inactive node (BNAND) at position 9 becomes active.

We can see that the phenotype at iteration 0 computes even-2 parity as follows. Denote the outputs of node i by z_i. The symbol $\oplus$ denotes the exclusive OR operation. When two or more Boolean arguments are side by side (as if being multiplied), it is assumed that the Boolean AND (BAND) operation is applied to the arguments (e.g. xy is equivalent to BAND (x,y)). An overbar represents inversion.

$$\begin{aligned}
z_0 &= x_1,\\
z_1 &= x_1,\\
z_2 &= x_0,\\
z_4 &= \mathrm{BNOR}(z_2,z_1) = \overline{x_0 + x_1} = \overline{x_0 x_1} = (1 \oplus x_0)(1 \oplus x_1),\\
z_5 &= \mathrm{BAND}(z_1,z_2) = x_1 x_0,\\
z_6 &= \mathrm{BOR}(z_5,z_4) = z_5 + z_4 = z_5 \oplus z_4 \oplus z_4 z_5.
\end{aligned} \tag{4.1}$$

Substituting for z_5 and z_4, expanding and then cancelling terms we obtain

$$\begin{aligned}
z_6 &= x_1 x_0 \oplus (1 \oplus x_0)(1 \oplus x_1) \oplus x_1 x_0 (1 \oplus x_0)(1 \oplus x_1),\\
z_6 &= x_1 x_0 \oplus 1 \oplus x_1 \oplus x_0 \oplus x_1 x_0 \oplus x_1 x_0 (1 \oplus x_1 \oplus x_0 \oplus x_1 x_0),\\
z_6 &= x_1 \oplus x_0 \oplus 1.
\end{aligned} \tag{4.2}$$

Thus $z_{16} = z_6$ is the even-2 parity function. When the eight duplicated nodes are inserted into the genotype just before position 2, they cause the activation of the BNAND node at position 9 in the new phenotype. This inverts the function computed by the eight duplicated nodes in the genotype. So the output of this block of nodes (denoted by A) is $x_2 \oplus x_1$, since the INPP functions return the inputs in descending order.

Now we turn our attention to the second iteration. When DUP inserts nodes 2–9 after itself, nodes 4–9 are shifted right (becoming nodes 12–17) in the second-iteration phenotype (enclosed in a box labeled B in Fig. 4.2). We now prove that this set of nodes carries out the exclusive OR of its input (emanating from the NAND node, which we call y) with the input variable (in this case x_1):

$$\begin{aligned}
z_{12} &= x_1,\\
z_{14} &= \mathrm{BNOR}(x_{12},y) = \mathrm{BNOR}(x_1,y) = (1 \oplus x_1)(1 \oplus y),\\
z_{15} &= \mathrm{BAND}(y,z_{12}) = \mathrm{BAND}(y,x_1) = x_1 y,\\
z_{16} &= \mathrm{BOR}(z_{15},z_{14}) = z_{15} + z_{14} = z_{15} \oplus z_{14} \oplus z_{14} z_{15},\\
z_{17} &= \mathrm{BNAND}(z_{16},z16) = z_{16} \oplus 1.
\end{aligned} \tag{4.3}$$

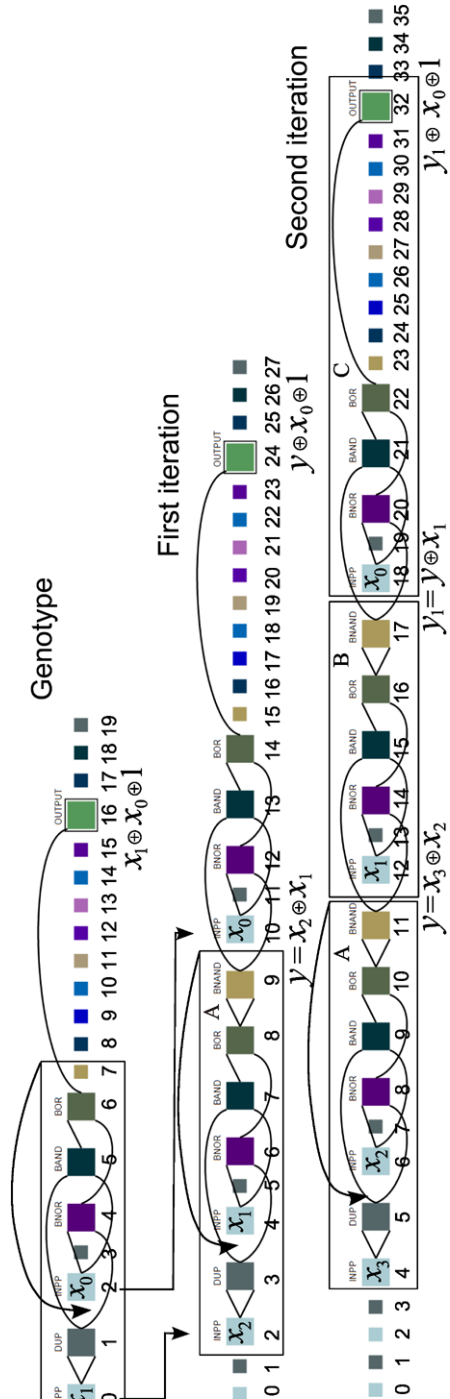

Fig. 4.2 An evolved genotype, iterated twice, that computes even-parity.

Substituting for z_{14} and z_{15} in z_{16} and then noting that in the last term, when $x_1 y$ multiplies $(1 \oplus x_1)$, we obtain $(x_1 y \oplus x_1 y)$, which is zero, we can simplify these equations as follows:

$$\begin{aligned} z_{16} &= x_1 y \oplus (1 \oplus x_1)(1 \oplus y) \oplus x_1 y (1 \oplus x_1)(1 \oplus y), \\ z_{16} &= x_1 y \oplus (1 \oplus x_1)(1 \oplus y) = x_1 y \oplus 1 \oplus x_1 \oplus y \oplus x_1 y, \\ z_{16} &= 1 \oplus x_1 \oplus y, \\ z_{17} &= x_1 \oplus y. \end{aligned} \tag{4.4}$$

Since we have seen that the nodes in section C compute the odd-parity of the supplied input y and the acquired input (by INPP), we find that at iteration 2 the phenotype computes $y_1 \oplus x_0 \oplus 1 = y \oplus x_1 \oplus x_0 \oplus 1 = x_3 \oplus x_2 \oplus x_1 \oplus x_0 \oplus 1$. This is the even-4-parity function. To construct a proof by induction, we will assume that for n inputs the phenotype computes even-n-parity.

The upper diagram in Fig. 4.3 shows the even-n-parity function E_n. This is the inductive hypothesis. We have already seen that the function enclosed in box A produces at the next iteration the two disconnected nodes and the function in A, $y = x_n \oplus x_{n-1}$, followed immediately by the function in box B, $y_{n-2} = y \oplus x_{n-2}$. Thus the new phenotype, E_{n+1}, generates the function

$$\begin{aligned} E_{n+1} &= y_{n-2} \oplus E_n \oplus x_{n-1} \oplus x_{n-2}, \\ E_{n+1} &= y \oplus x_{n-2} \oplus E_n \oplus x_{n-1} \oplus x_{n-2}, \\ E_{n+1} &= x_n \oplus x_{n-1} \oplus x_{n-2} \oplus E_n \oplus x_{n-1} \oplus x_{n-2}, \\ E_{n+1} &= x_n \oplus E_n. \end{aligned} \tag{4.5}$$

Thus the inductive hypothesis also applies to the $n+1$th iteration. We have already seen that at iteration 2, the form of the phenotype obeys the inductive hypothesis. Hence the general case is proved.

4.4.4 Why GP Cannot Solve General Parity Without Iteration

The number of test cases required to define even-n-parity is 2^n; from this, it follows in a simple manner that the number of test cases required to define all the even-parity functions from 2 to N is $4(2^{N-1} - 1)$. Clearly, it soon becomes impossible to evaluate the fitness of an evolved solution when N is too large.

SMCGP is able to address the scaling problem by evolving a program that can generate instances of arbitrary size. The use of function nodes (i.e. INP, INPP and SKIP) to acquire inputs and the ability to have multiple outputs (using the OUTPUT function) mean that as programs alter themselves they can use as many inputs as needed.

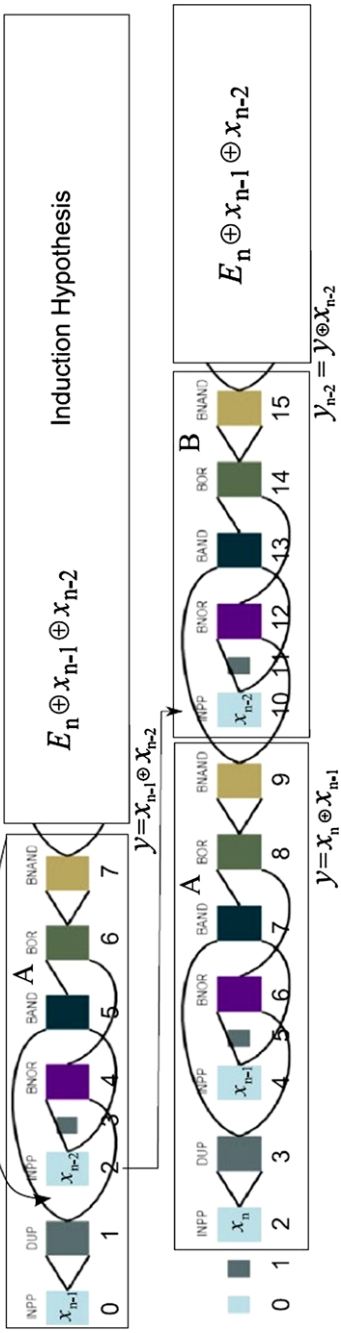

Fig. 4.3 The inductive hypothesis in diagrammatic form.

A statically sized CGP program can use only a fixed number of inputs – or, with the addition of functions such as INP, as many inputs as there are nodes in the graph. If the GP method does not have recursion or iteration (which is common), then it cannot acquire an arbitrary number of inputs. That is to say, it is always defined to have a specific number of inputs (i.e. even-6 or even-9). Since SMCGP has iteration built into the genotype–phenotype mapping, it can handle sequences of problems such as parity and many other problems [6, 7, 4].

It is worth contrasting evolved programs that have iteration in the form of feedback loops with SMCGP solutions. A program to do this would need to supply a bit-string and the number of bits in it. A general solution to even-n-parity can be obtained by reading in $n-1$ bits and EXORing them, and then, EXNORing with the last bit. In circuit terms, such a program would be a *sequential* circuit (it would have to be clocked). SMCGP solutions are different from this. The genotype is iterated n times to produce a *non-sequential circuit* that computes the parity. In other words, SMCGP produces a sequence of parallel parity circuits none of which have iteration in them. This has advantages and disadvantages. An advantage is that a parallel non-sequential circuit is faster than a sequential circuit with the same function. A disadvantage is that the size of the circuit is dependent on the number of inputs, whereas the size of a sequential program (or circuit) solving the problem is not dependent on the number of inputs.

Feedback could be introduced into CGP quite easily but it has only recently been given attention (see Sect. 2.9). One way would be to simply clock the feedback signals back to wherever they were connected (this could be done with a flip-flop). Clearly, feedback could also be introduced into SMCGP. It would be interesting to investigate the utility of this.

4.5 SM vs GP vs GA

Besides the many changes that we have made to accommodate a developmental approach to CGP by implementing self-modification in SMCGP, there is one other key issue where a substantial change has taken place: SMCGP trains on problems of different size and variable difficulty. Thus, it is not only the representation that has changed, but also the way we assign fitness to an individual in the course of evolution.

To understand this change, we need to review the historical changes that took place when genetic algorithms were first developed into genetic programming. Again, a key point at the time was the change of representation. From a bit-string representation of genetic algorithms (or a fixed number of parameters), the representation was changed to accommodate variable-length expression trees. This was a key innovation, allowing an individual to adjust to the problem size by growing (or sometimes shrinking) or more succinctly, by adapting its representation size.

As it was later pointed out and examined in detail by Langdon and Poli [13, 14], below a certain threshold of program size, an exact solution to a programming

problem is not possible at all. Above that critical threshold, however, there exist exact solutions to the problem, the more the larger the size of the program. This insight has relevance not only to genetic programming but also to all methods of automatic programming imaginable.

The other innovation of genetic programming was that evolution would be performed not on one particular problem configuration but on a multitude. In GP parlance, a set of fitness cases was used to determine the fitness of the program. This is in line with the functionality of a program which requires it to react to many situations, not just one. A program could perform relatively well by fulfilling certain fitness cases perfectly and others only weakly, or by performing equally well on all fitness cases (but without fulfilling any perfectly). The trade-off between these two types of solutions gave rise to generalization measurements, with a program doing the former being classified as a specialist, and a program doing the latter being classified as a generalist.

The main point here is that the measurement of fitness based on many fitness cases made the evolution of a more general solution possible in the first place. Without this input being offered to the system, generalization capabilities would have remained elusive to genetic programming, much as they are to a genetic algorithm, which usually performs an optimization for a particular solution only.

At the same time, a set of fitness cases constituted a problem for GP that required its representation to be flexible enough to adapt its complexity. Thus, the problem and its solution correspond to each other in the appropriate way.

In a similar vein, the progress envisioned with SMCGP depends on two interrelated innovations. On the one hand, the self-modification operations, together with the adjustment of the numbers of inputs and outputs, allow an individual solution not only to adapt to problem difficulty once and for all, but also to adapt continuously in the course of its existence. This is the representation aspect of the innovation.

The other aspect, however, is that during training and evolution, fitness cases from different problem dimensions and difficulties are used to determine the fitness of an individual. Thus, for example, in the case of the even-n-parity problem, evolution is fed with fitness cases for a number of problem dimensions $n = 2, 3, 4, 5, \ldots$. As we could see, if this is done in the proper way, SMCGP is able to extract structure from the fitness feedback that allows it to solve not only a single dimension of parity problems, e.g. $n = 5$, but to solve the problem for the case of general n.

In a way, SMCGP can generalize on a higher level: it can generalize from simple cases of a problem to more difficult ones, provided it has been shown in the course of evolution what this will entail in terms of developing a structure.

Many people have, for many years, maintained that GP is not really what we need since it would not be able to generalize appropriately, in the way a human can generalize from a set of sample problems to the general structure. This, however, is precisely what SMCGP can do, thanks to its corresponding features of SM operations, ability to choose input and output dimensions, and training with problem instances of varying difficulty.

Will SMCGP remain the only GP system that can do this? We doubt it. Sooner or later, self-modifying linear or tree GP systems will be designed, and the training

method of using problem instances of varying degree for fitness evaluation will be applied to those representations. In fact, it might be interesting to apply this training method to existing classical representations to study what effect the additional information offered in those examples has on a solution.

4.6 Implementing Incremental Fitness Functions

The fitness function described in Sect. 4.4.1 is typical of those used so far in SMCGP. But how does the incremental fitness function affect evolution?

In the experiment described in this section, we examined the even-parity problem but introduced a variation of the fitness function where the fitness function was no longer incremental.

We evolved circuits up to a certain number of test sets, and then checked to see if the solutions generalized to more test sets. For the incremental fitness function, the evaluation was aborted if the solution failed to correctly predict all bits in one test set. For the alternative fitness function, the number of test sets that the fitness function saw was fixed. This fitness function continued to evaluate iterations even if the previous iteration was unsuccessful.

From Table 4.6 we see that for smaller problems, the incremental fitness function requires more evaluations than does a fixed-sized fitness function. However, the incremental function soon becomes very efficient at finding solutions. The dip in the average number of evaluations shows where the solutions begin to exhibit generalization (see also Fig. 4.4).

Table 4.6 The average number of evaluations required to find a solution to the parity problem. Programs need to grow from two inputs through to the maximum number of inputs

Inputs	Fixed	Incremental
2	5,068	6,547
3	14,131	20,630
4	26,006	30,692
5	56,600	39,152
6	108,685	57,430
7	122,506	58,313
8	167,623	44,298
9	180,048	39,049
10	204,034	51,483

However, from the results in Table 4.7 and Fig. 4.5, we see that the fixed-sized fitness function is more likely to produce general solutions when they are evolved for larger-input problems. For a smaller maximum number of inputs, the incremental fitness function produces more general solutions, but still with low probability.

In addition to the increased number of evaluations required with the fixed-size fitness function, there is another penalty to pay in terms of the number of test sets

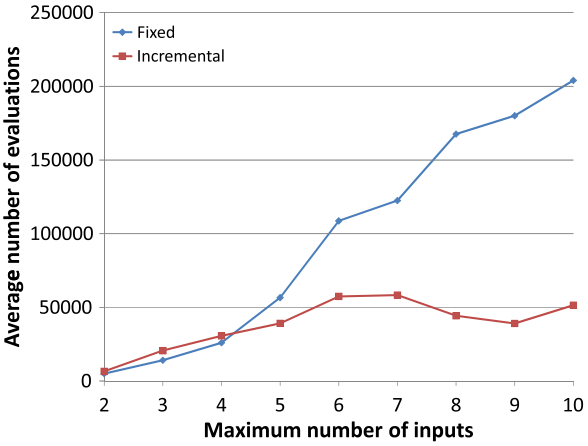

Fig. 4.4 The average number of evaluations required to find a solution to the parity problem. Programs need to grow from 2 inputs through to the maximum number of inputs.

Table 4.7 Percentage of solutions that, when evolved up to a given input size, continue to generalize to 20 inputs

Inputs	Fixed	Incremental
2	0	0
3	2	4
4	27	38
5	53	69
6	80	67
7	90	83
8	95	89
9	94	84
10	94	88

that need to be evaluated. Table 4.8 shows that the incremental fitness function requires far fewer test sets to be processed and in turn, we can deduce that far fewer test cases are required (as the earlier truth tables are small).

To conclude, we see that the incremental fitness function does not actively help generalization. However, the performance benefits in terms of the amount of the tests that need to be performed are likely to outweigh this. In future work, we will investigate if this behaviour holds for other problems.

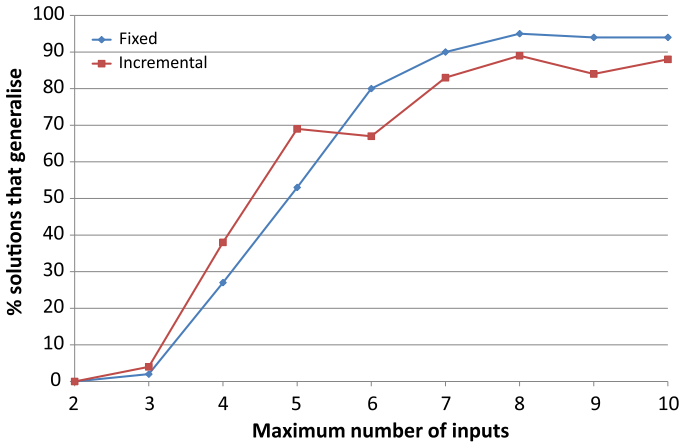

Fig. 4.5 Percentage of solutions that, when evolved up to a given input size, continue to generalize to 20 inputs.

Table 4.8 Average number of test sets evaluated when the solution is evolved up to a given input size

Inputs	Fixed	Incremental
2	5086	6565
3	28281	26969
4	78037	41912
5	226417	54829
6	543441	82278
7	735053	86203
8	1173376	65050
9	1440402	56945
10	1836328	78538

4.7 Conclusions

SMCGP has shown itself to be a capable and versatile extension to CGP. In the examples shown here and discussed elsewhere, we see that the addition of self-modification adds useful functionality and can be used to solve problems that a fixed-representation GP system cannot. Furthermore, we see that the implementation and representation are relatively straightforward. SMCGP can output human-readable programs that can be proved to behave in certain ways. Additionally, work has shown that the addition of self-modification does not harm the evolvability of CGP [4]. This gives us confidence that the SMCGP approach may be a suitable replacement for the classical CGP model.

Currently, a new version of SMCGP is under development that aims to simplify SMCGP (for example by having a more streamlined representation and function set) whilst at the same time increasing the developmental richness (by moving to a 2D representation). Early results on digital circuits [9] and mathematical results [8] are promising.

4.8 Acknowledgements

WB and SH gratefully acknowledge funding from Atlantic Canada's HPC network ACENET; from the Canadian Foundation of Innovation, New Opportunities Grant number 204503; and from NSERC under the Discovery Grant Program RGPIN 283304-07.

References

1. Banzhaf, W., Miller, J.F.: The Challenge of Complexity, chap. 11, pp. 243–260. Kluwer Academic (2004)
2. Harding, S., Miller, J.F., Banzhaf, W.: Self-modifying Cartesian Genetic Programming. In: Proc. Genetic and Evolutionary Computation Conference, pp. 1021–1028 (2007)
3. Harding, S., Miller, J.F., Banzhaf, W.: Development and Learning Using Self-modifying Cartesian Genetic Programming. In: Proc. Genetic and Evolutionary Computation Conference, pp. 699–706. ACM Press (2009)
4. Harding, S., Miller, J.F., Banzhaf, W.: Self-modifying Cartesian Genetic Programming: Fibonacci, Squares, Regression and Summing. In: Proc. European Conference on Genetic Programming, *LNCS*, vol. 5481, pp. 133–144 (2009)
5. Harding, S., Miller, J.F., Banzhaf, W.: Self-modifying Cartesian Genetic Programming: Parity. In: Proc. IEEE Congress on Evolutionary Computation, pp. 285–292 (2009)
6. Harding, S., Miller, J.F., Banzhaf, W.: Developments in Cartesian Genetic Programming: Self-modifying CGP. Genetic Programming and Evolvable Machines **11**, 397–439 (2010)
7. Harding, S., Miller, J.F., Banzhaf, W.: Self-modifying Cartesian Genetic Programming: Finding algorithms that calculate pi and e to arbitrary precision. In: Proc. Genetic and Evolutionary Computation Conference, pp. 579–586. ACM (2010)
8. Harding, S., Miller, J.F., Banzhaf, W.: SMCGP2: Finding Algorithms That Approximate Numerical Constants Using Quaternions and Complex Numbers. In: Proc. Genetic and Evolutionary Computation Conference Companion, pp. 197–198. ACM (2011)
9. Harding, S., Miller, J.F., Banzhaf, W.: SMCGP2: Self-modifying Cartesian Genetic Programming in Two Dimensions. In: Proc. Genetic and Evolutionary Computation Conference, pp. 1491–1498. ACM (2011)
10. Huelsbergen, L.: Finding General Solutions to the Parity Problem by Evolving Machine-Language Representations. In: J.R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D.B. Fogel, M.H. Garzon, D.E. Goldberg, H. Iba, R. Riolo (eds.) Proc. Conference on Genetic Programming, pp. 158–166. Morgan Kaufmann (1998)
11. Koza, J.R.: Genetic Programming: On the Programming of Computers by Natural Selection. MIT Press (1992)
12. Koza, J.R.: Genetic Programming II: Automatic Discovery of Reusable Programs. MIT Press (1994)

13. Langdon, W.B.: Scaling of program fitness spaces. Evolutionary Computation **7**(4), 399–428 (1999)
14. Poli, R., Langdon, W.B.: Foundations of Genetic Programming. Springer (2002)
15. Poli, R., Page, J.: Solving High-Order Boolean Parity Problems with Smooth Uniform Crossover, Sub-Machine Code GP and Demes. Genetic Programming and Evolvable Machines **1**(1–2), 37–56 (2000)
16. Schmidt, M., Lipson, H.: Distilling free-form natural laws from experimental data. Science **324**, 81–85 (2009)
17. Schmidt, M., Lipson, H.: Solving iterated functions using genetic programming. In: Proc. Conference Companion on Genetic and Evolutionary Computation, pp. 2149–2154. ACM (2009)
18. Spector, L., Clark, D.M., Lindsay, I., Barr, B., Klein, J.: Genetic programming for finite algebras. In: Proc. Genetic and Evolutionary Computation Conference, pp. 1291–1298. ACM (2008)
19. Spector, L., Robinson, A.: Genetic programming and autoconstructive evolution with the Push programming language. Genetic Programming and Evolvable Machines **3**, 7–40 (2002)
20. Streeter, M., Becker, L.A.: Automated Discovery of Numerical Approximation Formulae via Genetic Programming. Genetic Programming and Evolvable Machines **4**, 255–286 (2003)
21. Walker, J.A., Miller, J.F.: Investigating the Performance of Module Acquisition in Cartesian Genetic Programming. In: Proc. Genetic and Evolutionary Computation Conference, pp. 1649–1656. ACM (2005)
22. Walker, J.A., Miller, J.F.: Automatic Acquisition, Evolution and Re-use of Modules in Cartesian Genetic Programming. IEEE Transactions on Evolutionary Computation **12**, 397–417 (2008)
23. Wong, L.M.: Evolving Recursive Programs by Using Adaptive Grammar Based Genetic Programming. Genetic Programming and Evolvable Machines **6**, 421–455 (2005)
24. Wong, M.L., Leung, K.S.: Evolving Recursive Functions for the Even-Parity Problem Using Genetic Programming. In: P.J. Angeline, K.E. Kinnear, Jr. (eds.) Advances in Genetic Programming 2, chap. 11, pp. 221–240. MIT Press (1996)
25. Yu, T.: Hierachical Processing for Evolving Recursive and Modular Programs Using Higher Order Functions and Lambda Abstractions. Genetic Programming and Evolvable Machines **2**, 345–380 (2001)