Evolving High-Level Imperative Program Trees with Strongly Formed Genetic Programming

Tom Castle and Colin G. Johnson

School of Computing, University of Kent,
Canterbury, CT2 7NF, UK
{tc33,C.G.Johnson}@kent.ac.uk
http://www.kent.ac.uk

Abstract. We present a set of extensions to Montana's popular Strongly Typed Genetic Programming system that introduce constraints on the structure of program trees. It is demonstrated that these constraints can be used to evolve programs with a naturally imperative structure, using common high-level imperative language constructs such as loops. A set of three problems including factorial and the general even-n-parity problem are used to test the system. Experimental results are presented which show success rates and required computational effort that compare favourably against other systems on these problems, while providing support for this imperative structure.

1 Introduction

Evolving high-level imperative programs with genetic programming (GP) [1] is challenging. Programs are required to abide by complex structural rules just to be legal, and introducing commonly used constructs such as iteration bring further complexities to ensure that programs terminate and can be evaluated effectively. Because of these challenges, the GP research community has traditionally focused on the evolution of functional programs with a simple nested structure. However, the success of recent work using GP to perform tasks such as bug fixing commercial imperative programs [2], demonstrates the value of being able to evolve high-level imperative code. Modern imperative languages such as C/C++, Java and Python are widely used in the software development industry, so the evolution of code in this form is highly relevant for any application of GP in this area.

In this paper, we introduce Strongly Formed Genetic Programming (SFGP), a novel approach to constraining the structure of the program trees evolved with GP. SFGP extends previous work by Montana on Strongly Typed Genetic Programming [3] and combines it with constraints similar to those used by Koza in his work on *constrained syntactic structures* [1]. We demonstrate how the additional structural constraints of SFGP can be used to evolve high-level imperative code within a tree representation.

The rest of this paper is organised as follows. Section 2 discusses some of the previous work on evolving imperative programs. Section 3 then gives a detailed

description of how SFGP works. Section 4 describes the experiments that were conducted with the results presented and discussed in section 5. Finally, we conclude and summarise some of the research that this work leads on to in section 6.

2 Background

Some of the earliest attempts at evolving imperative programs were with a linear GP [4, 5] approach. In linear GP, programs are comprised of a sequence of either machine code or interpreted higher- level instructions. The instructions read from and write to registers. The main incentive for using linear GP is faster execution speed [6], since the instructions can often be executed directly on hardware with little or no interpretation.

Grammar-guided GP is another area that has prompted work into imperative GP. Grammars provide a mechanism for constraining source code that is generated within a valid predefined syntax. O'Neill and Ryan [7] evolved multi-line C programs in their Grammatical Evolution (GE) system to solve the Santa Fe ant trail problem. While shown to be successful at solving this problem, the use of the standard GE algorithm in evolving more complex imperative programs is difficult since it uses context-free grammars which lack the expressiveness to describe semantic constraints. Other authors [8–10] have described extensions to GE that use context-sensitive grammars, but none go as far as using the extensions to evolve imperative programs. More recently, Langdon [11] has demonstrated how a grammar-guided GP system can be used to evolve compilable C++ CUDA kernels. Other grammar-based approaches have made use of context-sensitive grammars such as DCTG- GP [12] and LOGENPRO [13] which uses logic grammars to induce programs in a range of languages, including imperative C programs.

Montana proposed Strongly Typed Genetic Programming (STGP) [3] to allow nodes to define data-type constraints on their inputs that will be satisfied by the algorithm. This removes the need for the nodes to satisfy the closure property [1]. STGP does not provide any explicit mechanism for restricting the structure of program trees as grammars do, but the data-typing constraints do go some way to providing the restrictions necessary for supporting an imperative structure. Imperative programs are inherently composed of sequences of statements which are to be executed in order. Koza [1] used `ProgN` functions to achieve something similar. However, as McGaughran and Zhang [14] observe, this approach may provide a sequential ordering but it does so without a control structure that corresponds to any standard imperative constructs. They go on to present their own system based on a method of chaining statements.

3 Strongly Formed Genetic Programming

In this section we describe our method of evolving programs with a naturally high-level imperative structure. We refer to the method by the name Strongly

Formed Genetic Programming (SFGP), since it extends Montana's STGP, with additional constraints to the structure or form of the program trees.

Let us first clarify the limitation of STGP with an example. STGP requires all non-terminals to define the required data-type of each of their inputs. But, no limitation can be imposed on which terminal or non-terminal will provide that input. The child node may be any terminal or non-terminal of the correct data-type. Consider a type of node that performs the variable assignment operation. Any non-trivial imperative program is likely to require such a node. This `Assignment` node will require two inputs: a variable, and a value of the same data-type to assign to that variable. STGP can easily constrain these two inputs to be of the same data-type, but requires additional constraints to limit the first child to be a `Variable` node, rather than any other node of that data-type. The same problem exists with constraining code-blocks to contain only statements, and loop constructs that require a variable to update with an index.

SFGP resolves this problem by introducing an additional requirement of non-terminal nodes; that they define both a *data-type* and a *node- type* for each argument. The node-type property of an argument is defined as being the required terminals or non-terminals that can be a child node at this point, which when evaluated will return a value of the specified data- type. This therefore provides an explicit constraint on the shape and structure of program trees. In the case of an integer `Assignment` node, this can be used to state that the first child should not only be of an integer data-type, but should also specifically be a `Variable` node. These constraints must then be satisfied throughout the evolutionary process with minor modifications to the initialisation, mutation and crossover operators, as described in the following sections.

3.1 Initialisation

SFGP uses a grow initialisation procedure to construct random program trees, where each node is selected at random from those with a compatible data-type and node-type required by its parent (or the problem itself for the root node). Montana's grow initialisation operator made use of lookup tables to check whether a data-type is valid at some depth, but the addition of a second constraint excessively complicates these tables. The alternative is to allow the algorithm to backtrack when no valid nodes are possible for the required constraints. At each step, if no valid nodes are possible within the available depth, then the function returns an error, and if the construction of a subtree fails with an error then an alternative node is chosen and a new subtree generated at that point. The algorithm ensures that all program trees that are generated satisfy all data-type and node-type limitations and that each tree is within the *maximum-depth* parameter.

Pseudo-code for the grow initialisation algorithm is listed in algorithm 1. The `GenerateTree` function is initially called with a `data_type` parameter that is the required return type for the problem and a `node_type` parameter which defines the node-type required for the root of the program tree. On all problems in this paper, a root node of `SubRoutine` is used, which is intended to model a

Algorithm 1 Initialisation procedure, where dt, nt and $depth$ are the required data-type, node-type and maximum depth. The $filterNodes(S, dt, nt, depth)$ function is defined to return a set comprised of only those nodes in S with the given data-type and node-type, and with non-terminals removed if $depth = 0$.

```
 1: function GenerateTree(dt, nt, depth)
 2:     V ← filterNodes(S, dt, nt, depth)
 3:     while V not empty do
 4:         r ← removeRandom(V)
 5:         for i ← 0 to arity(r) do
 6:             dti ← required data-type for ith child
 7:             nti ← required node-type for ith child
 8:             subtree ← generateTree(dti, nti, depth − 1)
 9:             if subtree ≠ err then
10:                 attach subtree as ith child
11:             else
12:                 break and continue while
13:             end if
14:         end for
15:         return r                              ▷ Valid subtree complete
16:     end while
17:     return err                                ▷ No valid subtrees exist
18: end function
```

module of code such as a function or method. This means that all programs that are generated have the same basic imperative structure, shown in figure 1. The `SubRoutine` node requires two children: a `CodeBlock` with a void data-type and a `Variable` with the same data-type as the subroutine. Nodes with a void data-type do not return a value. When evaluated, a subroutine's code-block, which is a list of some predefined number of statements, is first executed and then the value of the variable is returned as the result of the subroutine.

3.2 Mutation

Our mutation operator employs the initialisation algorithm to grow new subtrees of the same data-type and node-type as an existing randomly selected node in a program tree. This node is then replaced with the newly generated subtree. The initialisation procedure is able to generate trees within a given maximum depth, so replacement subtrees are generated to be no deeper than the maximum depth parameter, minus the depth of the mutation point. Assuming the set of available nodes is unchanged, then it should always be possible to generate a legal replacement subtree for any existing node, but it is possible that the replacement is syntactically or semantically identical to the existing subtree. It is possible that this could lead to a high degree of neutral mutation if the syntax contains little variety.

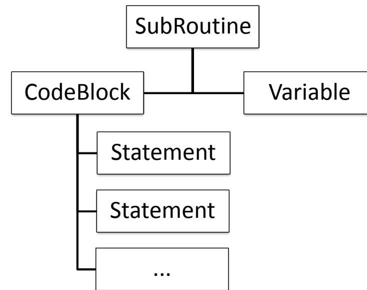


Fig. 1: The imperative structure of all program trees. In this paper, `CodeBlocks` with exactly 3 statement arguments were used. This was an arbitrary choice and an alternative size may produce different results. Evaluation executes each of the code-block's statements, before returning the value of the sub-routine's variable.

3.3 Crossover

The subtree crossover operator has been modified to maintain the node-type constraint while exchanging genetic material between two program trees. A node is selected at random in one of the programs. A second node is chosen at random from those nodes in the other program that are of the same data- type and node-type as the first node. The subtrees rooted at these two selected nodes are then exchanged. Those resultant child programs that have depths that exceed the maximum depth parameter are discarded.

3.4 Polymorphism

When implemented with an object-oriented approach, SFGP is able to support a simple form of polymorphism for both the data-type and node-type constraints. In figure 1 the `CodeBlock` node is shown to have a sequence of children with a `Statement` node-type. In an object-oriented system, this can be interpreted as any object that is an instance of the `Statement` class, or any sub-class. Nodes such as `Assignment`, `IfStatement` and `ForLoop` may then be implemented as sub-classes of `Statement` and may all appear in this position. In fact, it makes little sense to create a node of the type `Statement` itself, it is merely used to maintain the hierarchy of node-types. We refer to such node-types as *abstract* node-types. `Expression` is another abstract node-type that is used frequently. Data-type constraints can make use of the same polymorphic properties. If `Integer` and `Float` are both sub-classes of a class called `Number`, then either may appear where a required data-type of `Number` is specified. Note that the object-oriented approach we refer to here is a property of the implementation, rather than of the evolved programs, which are not themselves object-oriented.

3.5 Syntax

In this section we itemise the list of nodes that are used in the rest of this paper, along with the required data-type and node-types for their arguments.

- `SubRoutine` - requires a void `CodeBlock` and a `Variable` of the same data-type as the sub-routine. On evaluation, the code-block is evaluated and then the value of the variable is returned as the result. In all our experiments, `SubRoutine` is defined as the required root node-type.
- `CodeBlock` - is defined to require a fixed number of child nodes (3 used in all cases within this paper) of a `Statement` node-type and a void data-type. On evalutation, each child will be evaluated in sequence. `CodeBlock` has a void data-type and so does not return a value.

The following node-types are all subtypes of the abstract `Statement` node-type. All statements have a void data-type and so do not return any value.

- `Loop` - requires two children: an integer `Expression` and a `CodeBlock`. The expression is evaluated to provide a number of iterations to perform, and the code-block is evaluated the specified number of times. To maintain reasonable evaluation times, the number of iterations is capped at 100. No variables are manipulated by this loop construct.
- `ForLoop` - requires an integer `Variable`, which is updated with the index on each iteration from 1 to an upper bound; an integer `Expression` which is evaluated once to supply the upper bound (capped at 100 iterations) and a `CodeBlock` which is evaluated once per iteration.
- `ForEachLoop` - requires three children: a `Variable` of the element data-type, an `Expression` of some pre-defined array data-type and a `CodeBlock`. The code-block is evaluated once for each element of the array obtained by evaluating the expression argument. For each iteration, the value of the element is assigned to the variable.
- `IfStatement` - requires one boolean `Expression` and one `CodeBlock`. The code-block is conditionally evaluated only if the expression evaluates to true.
- `Assignment` - requires one `Variable` and one `Expression` input. Both inputs are required to have the same data- type specified on construction. On evaluation the expression is evaluated, and the result is assigned as the value of the variable.

The following node-types are all subtypes of the abstract `Expression` node-type. All expressions have non-void data-types.

- `Add`, `Subtract`, `Multiply` - require two integer `Expression` children each, with the integer result of the arithmetic operation returned.
- `And`, `Or`, `Not` - require two, two and one boolean `Expression` children respectively, with the boolean result returned.
- `Literal` - holds a fixed literal value of a given data-type.
- `Variable` - holds a value of a given data-type which may be modified (by assignment) throughout evaluation. The data-type of a variable is fixed at construction.

4 Experiments

A series of experiments were carried out to demonstrate SFGP's ability to generate programs composed of standard programming constructs, within the imperative structure enfored by our constraints. All experiments were carried out using the EpochX evolutionary framework [15], with our own extension in which the representation and operators were implemented as described in section 3. 500 runs were performed on each of three problems: factorial, Fibonacci and even-n-parity. These problems were chosen because they require the use of essential programming constructs such as loops and arrays and have also been used numerous times previously in the literature [16–18]. A maximum of 50 generations and a population of 500 were used on all problems. The subtree crossover and subtree mutation operators were chosen from with probabilities 0.9 and 0.1 respectively and tournament selection was used with a tournament size of 7. All other control parameters used are outlined in tables 1, 2 and 3.

4.1 Factorial

The program to be evolved here is an implementation of the factorial function. One input is provided, which is the integer variable `i`, where the ith element of the sequence is the expected result. The first 20 elements of the sequence were used to evaluate the quality of solutions, with a normalised sum of the error used as an individual's fitness score. The fitness function is defined in (1), where n is the size of the training set, i is the ith training case, $f(i)$ is the correct result for training case i, and $g(i)$ is the estimated result for training case i returned by the program under evaluation. Each individual which successfully handles all training inputs is tested for generalisation using a test set consisting of elements 21 to 50 of the sequence.

$$Fitness = \sum_{i=0}^{n} \frac{|f(i) - g(i)|}{|f(i)| + |g(i)|} \tag{1}$$

Table 1: Parameter tableau for the factorial problem

Root data-type:	`Integer`
Root node-type:	`SubRoutine`
Max. depth:	6
Non-terminals:	`SubRoutine`, `CodeBlock`, `ForLoop`, `Assignment`, `Add`, `Subtract`, `Multiply`
Terminals:	i, $loopVar$[1], 1 (integer `Literal`)

[1] The additional input *loopVar* is an integer variable, provided specifically for use by the `ForLoop` construct to contain the iteration index. Its initial value is 0.

4.2 Fibonacci

The Fibonacci problem was posed in a similar form as factorial, with an integer variable input `i` and an expected output which is the ith element of the Fibonacci sequence. Two further inputs were also provided in the form of variables containing the value of the first two elements of the sequence; 0 and 1. The same function (1) was also used to determine an individual's fitness, with the training inputs comprised of the first 20 elements of the Fibonacci sequence. A test set made up of elements 21 to 50 of the sequence were used to test the generalisation of successful programs.

Table 2: Parameter tableau for the Fibonacci problem

Root data-type:	`Integer`
Root node-type:	`SubRoutine`
Max. depth:	6
Non-terminals:	`SubRoutine`, `CodeBlock`, `Loop`, `Assignment`, `Add`, `Subtract`
Terminals:	i, $i0$, $i1$

4.3 Even-n-Parity

The boolean parity problems are widely used as a benchmark task in the GP literature [1, 5, 19]. However, they have only occasionally been tackled in the general form; for all values of `n` [18, 20]. A program which successfully solves the even-n-parity problem, must receive as input an array of booleans, arr, of any length and must return a boolean `true` value if an even number of the elements are `true`, otherwise it must return `false`. All possible inputs to the 3-bit even-parity problem were used as the training data, as used by Wong and Leung [18]. The fitness of an individual was then a simple count of how many of the 8 inputs were incorrectly classified. A test set consisting of all possible input arrays of lengths 4 to 10 was used to test the generalisation of solutions that successfully solved the training cases.

Table 3: Parameter tableau for the even-n parity problem

Root data-type:	`Boolean`
Root node-type:	`SubRoutine`
Max. depth:	8
Non-terminals:	`SubRoutine`, `CodeBlock`, `ForEachLoop`, `IfStatement`, `Assignment`, `And`, `Or`, `Not`
Terminals:	arr, $boolVar1$, $boolVar2$[2]

[2] *boolVar1* and *boolVar2* are boolean variables for use by the `ForEachLoop` and root `SubRoutine` constructs. Their initial values were arbitrarily set as `false`.

5 Results

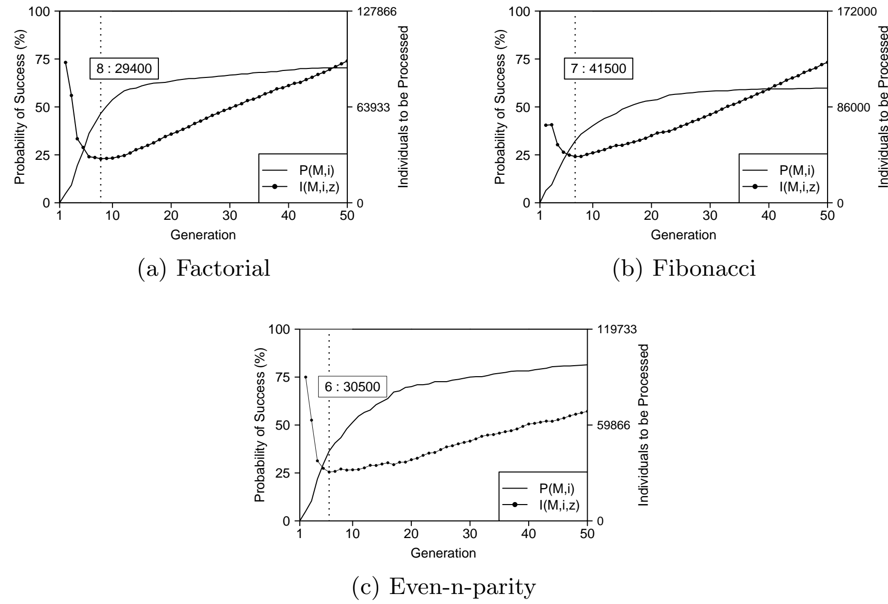
Table 4 lists a summary of the results, with the success rates and generalisability of the solutions that were discovered in the experiments. The table also describes the computational effort that was required to solve each problem, with the related performance curves displayed in figures 2a, 2b and 2c. The required computational effort was calculated in the manner of Koza [1] to be the number of individuals that must be processed to guarentee a solution with 99% confidence.

Previous attempts at evolving recursive structures that can generate the Fibonacci sequence include Harding et al [16], who used Self-Modifying Cartesian GP to generate both the first 12 and first 50 elements of the sequence with success rates up to 90.8% and up to 94.5% of those able to generalise to 74 elements of the sequence. In [21], a Linear GP system was used to achieve success rates up to 92%, and which generalised in 78% of cases. Other approaches have been less successful. Agapitos and Lucas used Object Oriented GP[17] to get success rates up to 25% on the first 10 elements of the sequence, but required a minimum computational effort of 2 million. Their success rates were slightly improved on the factorial problem, on which they report a 74% success rate and a minimum effort of 600,000. However, they note that their approach, which relies on Java's Reflection mechanism, is computationally expensive.

The even parity problems are considered to be difficult problems for GP to solve [20]. Koza's experiments required 1,276,000 individuals to be processed to yield a solution to just the 4-bit version of the problem and was unable to solve the problem with any higher number of bits without the use of automatic functions. In contrast, SFGP produced potential solutions with just 30,500 individuals processed, 81.4% of which were able to solve the general form of the problem including the 4-bit version. Other research has tackled the general even-n-parity problem. Agapitos and Lucas[17] required 680,000 individuals to be processed, where they used all the even-2-parity and even-3-parity problems as training data. In [18], Wong and Leung used just the even-3-parity problems as training inputs and reported their minimum computational effort as 220,000.

Table 4: Results summary, where *Train%* is the proportion of runs that produced at least one solution for all training cases and *Test%* is the proportion that produced a solution that generalised to solve all test cases. The confidence interval for the computational effort is calculated using the Wilson 'score' method [22]. The *Evals* column shows the number of program evaluations required to find a solution, which is a product of the effort and number of training cases.

	Train %	Test %	Effort	95% CI	Evals
Factorial	70.8	70.6	29,400	25,800 - 33,500	588,000
Fibonacci	61.6	59.8	41,500	35,600 - 48,500	830,000
Even-n-Parity	91.6	81.4	30,500	26,400 - 35,400	244,000



(a) Factorial

(b) Fibonacci

(c) Even-n-parity

Fig. 2: Performance curves, where $P(M, i)$ is the success rate and $I(M, i, z)$ is the number of individuals to process to find a solution with 99% confidence.

The primary reason for SFGP's greater performance on even-n-parity is likely to be due to the use of the `ForEachLoop` node, which encapsulates the necessary behaviour of performing an operation on each element of the array. The other studies mentioned relied on complex recursive structures developing through evolution.

5.1 Example solution

As an example, one typical solution to the factorial problem that was found by SFGP is displayed below:

```
public long getFactorial(long i) {
    loopVar = loopVar;
    long x = i;
    loopVar = 1L;
    for (long y = 1L; y <= x; x++, loopVar = x) {
        loopVar = (loopVar * i);
        i = loopVar;
        loopVar = loopVar;
    }
    i = i;
    return i;
}
```

This example program is expressed using Java syntax, but the abstract syntax trees generated by SFGP can be interpreted as programs in the syntax of any programming language that supports the imperative constructs used. This program has not undergone any post-processing, and contains statements such as $i = i$ which will have no impact on the result. These could easily be identified and removed by static analysis tools. Note that the loop structure contains the necessary infrastructure to ensure the index variable is updated but the bounds remain immutable, as defined by the `ForLoop` node that it represents.

6 Conclusions

In the course of this paper, we have introduced Strongly Formed Genetic Programming (SFGP) and demonstrated how it can be used to constrain the structure of program trees. In particular, we have shown that it is able to constrain evolved program trees to a more natural high-level imperative structure and make use of some standard imperative language constructs such as loops. The program trees that are evolved using this system may be easily expressed in the syntax of modern imperative programming languages. The results of using this imperative structure, that have been presented, compare very favourably with existing systems on these problems.

One current limitation with SFGP, that we would like to address in future work, is the lack of support for generic functions. Other possible future work includes investigating the impact of the arbitrary settings that have been used in this paper, such as the iteration bound on loop constructs and the fixed number of statements in a code-block. Support for imperative concepts such as variable declarations and recursion could also be of some value.

References

1. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
2. Weimer, W., Nguyen, T., Le Goues, C., Forrest, S.: Automatically finding patches using genetic programming. In Fickas, S., et al., eds.: ICSE 2009, Vancouver (2009) 364–374
3. Montana, D.J.: Strongly typed genetic programming. Evolutionary Computation **3** (1995) 199–230
4. Nordin, P.: A compiling genetic programming system that directly manipulates the machine code. In Kinnear, Jr., K.E., ed.: Advances in Genetic Programming. MIT Press (1994) 311–331
5. Brameier, M., Banzhaf, W.: Linear Genetic Programming. Number XVI in Genetic and Evolutionary Computation. Springer (2007)
6. Poli, R., Langdon, W.B., McPhee, N.F.: A Field Guide to Genetic Programing. Lulu.com (2008)
7. O'Neill, M., Ryan, C.: Evolving multi-line compilable C programs. In Poli, R., Nordin, P., Langdon, W.B., Fogarty, T.C., eds.: EuroGP 1999. Volume 1598 of LNCS., Goteborg, Springer (1999) 83–92

8. Cleary, R., O'Neill, M.: An attribute grammar decoder for the 01 multiconstrained knapsack problem. In Raidl, G.R., Gottlieb, J., eds.: EvoCOP 2005. Volume 3448 of LNCS., Lausanne, Springer (2005) 34–45
9. de la Cruz, M., Ortega de la Puente, A., Alfonseca, M.: Attribute grammar evolution. In Mira, J., Álvarez, J.R., eds.: IWINAC 2005, Part II. Volume 3562 of LNCS., Las Palmas, Springer (2005) 182–191
10. Ortega, A., de la Cruz, M., Alfonseca, M.: Christiansen grammar evolution: Grammatical evolution with semantics. IEEE Transactions on Evolutionary Computation **11** (2007) 77–90
11. Langdon, W.B., Harman, M.: Evolving a CUDA kernel from an nVidia template. In Sobrevilla, P., et al., eds.: 2010 IEEE World Congress on Computational Intelligence, Barcelona, IEEE Press (2010) 2376–2383
12. Ross, B.J.: Logic-based genetic programming with definite clause translation grammars. In Banzhaf, W., Daida, J.M., Eiben, A.E., Garzon, M.H., Honavar, V., Jakiela, M.J., Smith, R.E., eds.: GECCO 1999. Volume 2., Orlando, Morgan Kaufmann (1999) 1236
13. Wong, M.L., Leung, K.S.: Combining genetic programming and inductive logic programming using logic grammars. In: 1995 IEEE Conference on Evolutionary Computation. Volume 2., Perth, IEEE Press (1995) 733–736
14. McGaughran, D., Zhang, M.: Evolving more representative programs with genetic programming. International Journal of Software Engineering and Knowledge Engineering **19** (2009) 1–22
15. Castle, T., Beadle, L.: Epochx: genetic programming software for research. `http://www.epochx.org` (2007)
16. Harding, S., Miller, J., Banzhaf, W.: Self modifying cartesian genetic programming: Fibonacci, squares, regression and summing. In Vanneschi, L., Gustafson, S., Moraglio, A., Falco, I.D., Ebner, M., eds.: EuroGP 2009. Volume 5481 of LNCS., Tübingen, Springer (2009) 133–144
17. Agapitos, A., Lucas, S.M.: Learning recursive functions with object oriented genetic programming. In Collet, P., Tomassini, M., Ebner, M., Gustafson, S., Ekrt, A., eds.: EuroGP 2006. Volume 3905 of LNCS., Budapest, Springer (2006) 166–177
18. Wong, M.L., Leung, K.S.: Evolving recursive functions for the even-parity problem using genetic programming. In Angeline, P.J., Kinnear, Jr., K.E., eds.: Advances in Genetic Programming 2. MIT Press, Cambridge (1996) 221–240
19. Castle, T., Johnson, C.G.: Positional effect of crossover and mutation in grammatical evolution. In Esparcia-Alcazar, A.I., Ekart, A., Silva, S., Dignum, S., Uyar, S., eds.: EuroGP 2010. Volume 6021 of LNCS., Istanbul, Springer (2010) 26–37
20. Harding, S., Miller, J.F., Banzhaf, W.: Self modifying cartesian genetic programming: Parity. In Tyrrell, A., et al., eds.: 2009 IEEE Congress on Evolutionary Computation, Trondheim, IEEE Press (2009) 285–292
21. Wilson, G., Heywood, M.: Learning recursive programs with cooperative coevolution of genetic code mapping and genotype. In Thierens, D., et al., eds.: GECCO 2007. Volume 1., London, ACM Press (2007) 1053–1061
22. Walker, M., Edwards, H., Messom, C.: Confidence intervals for computational effort comparisons. In Ebner, M., O'Neill, M., Ekárt, A., Vanneschi, L., Esparcia-Alcázar, A., eds.: EuroGP 2007. Volume 4445 of LNCS., Valencia, Springer (2007) 23–32