Evolving Program Trees with Limited Scope Variable Declarations

Tom Castle
School of Computing
University of Kent
Canterbury, Kent CT2 7NF
Email: tc33@kent.ac.uk

Colin G. Johnson
School of Computing
University of Kent
Canterbury, Kent CT2 7NF
Email: C.G.Johnson@kent.ac.uk

***Abstract*—Variables are a fundamental component of computer programs. However, rarely has the construction of new variables been left to the evolutionary process of a tree-based Genetic Programming system. We present a series of modifications to an existing GP approach to allow the evolution of high-level imperative programs with limited scope variables. We make use of several new program constructs made possible by the modifications and experimentally compare their use. Our results suggest the impact of variable declarations is problem dependent, but can potentially improve performance. It is proposed that the use of variable declarations can reduce the degree of insight required into potential solutions.**

I. Introduction

Variables are a fundamental component of computer programs. However, rarely has the power of constructing new variables been bestowed upon the evolutionary process of a tree-based Genetic Programming (GP) [1] system. Without variable declarations, all variables must be supplied as inputs to the system, including any auxiliary variables required for the computation process that are not part of the specified inputs or outputs. With complex programs, this can require a considerable degree of insight into the solution space. By supporting the evolution of variable declarations, the aim is to lighten this burden without excessively degrading performance.

It has previously been described how the Strongly Formed Genetic Programming (SFGP) variant of GP can be used to enforce a high-level imperative structure on evolved program trees [2]. In this paper, we present a series of modifications to SFGP that allow it to support node-types that can declare new limited scope variables. Limited scope variables are commonly found in modern high-level imperative programming languages, but are particularly challenging to incorporate into an evolutionary system. The difficulty is that each variable must not be used prior to being declared, nor beyond the extent of its scope. Neglecting the limited scope aspect of variable declarations may simplify the problem. But, this is inconsistent with the way local variables are used by human programmers, and produces programs reliant on global variables [3].

One of the frequently mentioned issues with genetic programming is the difficulty in evolving iteration or recursion [4], [5]. If a mechanism for supporting variable declarations is used, iterative constructs that more closely resemble those used in high-level imperative programming languages are simple to implement. These may supply indices or elements through variables that they declare. Such constructs are commonly used in human written code, and it seems likely that they could help to expand the range and scale of problems GP can be applied to.

The rest of this paper will be structured as follows. Section II will discuss some of the related work in the literature, including a brief summary of variable use in GP. This will be followed by a description of the SFGP system that is to be modified to support variable declarations. These modifications will be outlined in Section IV, along with a list of new constructs that can be supported. A series of experiments to examine the effect of these constructs on the evolutionary performance will be given in Sections V and VI. Finally, we conclude and summarise some future work.

II. Related Work

Variables are widely used in applications of GP for a variety of purposes. The inputs for programs in a GP population are typically supplied using variables, with the set of inputs defined by the GP practitioner and would normally be the same for all programs in a population. There is often no facility for the value of these variables to be altered. However, Koza [1, Chapter 18.2] did propose a mechanism for assigning the value of a global variable using a `SET-SV` operator. He suggested that the use of a settable variable like this was beneficial for the evolution of building blocks, since the variable provided a way of labelling a useful computation so that it could be used elsewhere in the program. Koza's approach not only treated all variables as global, but also required them to be defined in advance; no variable declarations here.

Linear GP variants [6], [7] commonly make use of defined memory registers which can be both assigned to and have values retrieved from them. The number of available registers is defined in advance to include registers for each input, plus additional registers for facilitating calculations. Brameier and Banzhaf [7, Chapter 2.1] make the point that it is important for a sufficient number of registers to be provided to avoid valuable information being overwritten. However, too many

registers may spread the computation too widely, and make it difficult to build a solution. As Oltean and Grosan [8] put it, "The number of supplementary registers depends on the complexity of the expression being discovered. An inappropriate choice can have disastrous effects on the program being evolved".

Stack based GP systems [9], [10] provide an alternative approach to memory, where the result of expressions are pushed onto a stack and popped off as inputs are required. As such, they are able to support an expandable memory allocation (within some reasonable bounds). PushGP [10] evolves programs in the specially designed Push programming language. Push provides a `NAME` data type, which maintains its own stack of variable labels, upon which values may be pushed by a program to define a new variable.

There has been some limited use of variable declarations with tree based GP approaches. The authors of OOGP [11] imply their existence by stating that "new local variables may occur within block statements". But they fail to give any additional details. A far more thorough explanation is given by Kirshenbaum [12], in his work with *statically scoped local variables*. He describes a method for supporting Lisp's `LET` expression, which is able to define variable bindings with limited scope. This is achieved by adding each `LET` expression's bindings to the set of available operators for each of the expression's subtrees as they are generated.

We take a similar approach to Kirshenbaum, but must deal with a slightly more complicated scenario, to cater for an imperative structure based on statements and blocks. The scope of a variable in an imperative program should not just descend into the operator's arguments, but should also be accessible to sibling operators (for example, statements following a declaration, within the same block).

III. Strongly Formed Genetic Programming

Strongly Formed Genetic Programming (SFGP) [2] is a technique for evolving program trees that conform to strict structural constraints. It extends Montana's Strongly Typed Genetic Programming (STGP) [13] system, so also includes strong data-typing restrictions. The authors demonstrate that it can be used to enforce an imperative structure equivalent to many popular high-level imperative programming languages, and make use of some standard programming constructs such as *for loops* and *for-each loops*.

STGP imposes a requirement upon all terminals and non-terminals to define the data-type of their return value and a further requirement of all non-terminals to define the required data-type of each of their arguments. SFGP has the same requirements, with one addition: all non-terminals must also define the required *node-type* for each of their arguments. The node-type property of an argument is defined as being the required terminal or non-terminal that can be a child node at this point, which when evaluated will return the value of the specified data-type. Both *abstract* and *concrete* node-types are used, where the possible concrete node-types are all those found in the terminal and non-terminal sets, and the abstract node-types are those that exist only in the inheritance hierarchy of the concrete nodes[1], so cannot themselves be instantiated. The use of abstract node-types allow multiple concrete node-types to be supported for an argument. For example, a `Statement` abstract node-type could be satisfied with an `Assignment`, an `IfStatement`, or a `Declaration`. The initialisation and genetic operators of SFGP are designed to ensure all of these constraints are satisfied.

The structural constraints make it possible to enforce an imperative structure comprised of statements and blocks. They also enable some additional programming constructs to be represented, that cannot be supported with just STGP. A generalised `Assignment` statement for setting the value of variables is possible, rather than requiring one `SET-VAR-`x operator per variable x. This is possible because the `Assignment` operator may define one of its arguments to be a variable, whereas in STGP only the data-type could be restricted. For the same reason, more useful loop constructs are enabled that assign the value of their index or the element being processed to an available variable on each iteration. One of the limitations that this paper seeks to address, is that a variable of the correct data-type must already exist and will then be hijacked for the loop's own purpose.

IV. Algorithm Modifications

There are two forms of variable declaration that we wish to support, each requiring different scope, consistent with modern imperative languages such as C/C++ and Java:

- Standard declarations create a new variable and assign it a value according to some expression. The variable's scope extends from the statement following the declaration, up to the end of the block the declaration was contained within. The variable must not be in scope for the declaration's own subtrees, but should be available at any level of nesting for the following statements, up to the point it is removed from scope.
- Some more advanced statement types, such as loops, may declare variables for use only within the body of a child block. This is the case for loop constructs that declare a new variable, which is updated on each iteration with the index or element. These variables should be available at any level of nesting within the loop statement that declared them, but not beyond.

To support these types of declaration, we introduce *syntax updates*. A syntax update is an opportunity for a node to modify the terminal and non-terminal sets. These syntax updates can be applied as the initialisation procedure progresses in order to change the available syntax for the construction of a node's subtrees or any following nodes. A syntax update may involve the addition or removal of nodes from the syntax. Each node is able to define $arity+1$ syntax updates, which when the tree is traversed depth-first, are applied before and after each of its child nodes are processed. Figure 1 illustrates this, with the dotted branches indicating the points of each syntax update,

[1] where an object-oriented implementation is assumed.

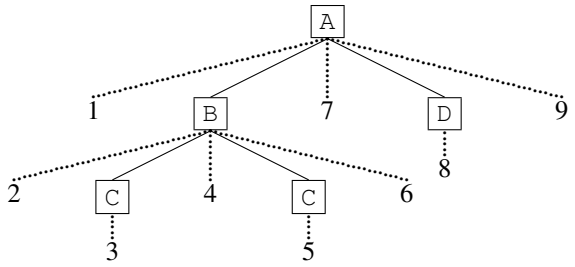


Fig. 1: Example illustrating the position and order of syntax updates. The syntax updates are indicated by dotted branches.

which are labelled with the order they would be applied. In this example, the syntax updates 2, 4 and 6 are all defined by the `B` node. Syntax updates 2 and 4 could be used to add variables, for use in one or both of its child subtrees. These variables could be removed in the syntax update labelled 6, which would restrict their scope to the node's child subtrees. Syntax update 6 could also be used to define variables to be available for successive nodes, such as node `D`.

The following sections describe the modifications that are necessary to use these syntax updates to evolve programs with limited scope variable declarations.

A. 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). The only modification necessary to support variable declarations, is for the syntax updates for each node to be applied as the tree is constructed. High-level pseudocode for the initialisation procedure is shown in algorithm 1. In the example, in Figure 1, the initialisation procedure would have applied the root's first syntax update, labelled 1, prior to selecting which node should be set as its first child. Any modifications made to the syntax in that first syntax update, would have influenced which nodes were available to be selected from. The initialisation would have continued down the tree, filling the nodes in a depth-first manner, with each syntax update applied in the order indicated. This ensures that each node that is selected, is chosen from only those that should be available at that point.

B. Mutation

A program tree undergoing SFGP's subtree mutation has a node randomly selected and replaced with a newly generated subtree with a compatible data-type and node-type. To support the dynamic syntax, the randomly generated subtree must be constructed from the available syntax at the mutation point, including any additional in-scope variables that have been declared prior to that point. The available syntax at the mutation point can be easily obtained by performing a partial traversal of the program tree, only up to the mutation point, with each node's syntax updates applied. The new subtree for the mutation can then be constructed from the updated syntax using the initialisation procedure.

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 $depth = 0$. The $updateSyntax(S, r, i)$ function performs the task of updating the available syntax, S, as defined for the ith position of the node-type r.

```
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:              S ← updateSyntax(S, r, i)
7:              dti ← required data-type for ith child
8:              nti ← required node-type for ith child
9:              subtree ← generateTree(dti, nti, depth − 1)
10:             if subtree ≠ err then
11:                 attach subtree as ith child
12:             else
13:                 break and continue while
14:             end if
15:         end for
16:         S ← updateSyntax(S, r, arity(r))
17:         return r                    ▷ Valid subtree complete
18:     end while
19:     return err                      ▷ No valid subtrees exist
20: end function
```

There is one further problem that needs to be overcome. No restrictions are in place on which subtree may be selected for replacement by the mutation operator. So, a node which performs a variable declaration could be replaced, potentially leaving *dangling* variables. A dangling variable, in this case, is a use of a variable without an associated declaration. To resolve this issue, a repair operation is performed, which is described in Section IV-D.

C. Crossover

Subtree crossover in SFGP operates on two program trees. A node is randomly selected in one program tree and the subtree rooted at that node is swapped with a subtree from the other program tree, selected at random from those with a compatible data-type and node-type. The dynamic syntax requires no modifications to this basic practice. However, there are two specific scenarios that need to be handled for variable declarations to be supported. (1) As with mutation, the subtree that is removed may contain the declaration for variables that are used elsewhere in the program tree. These would become dangling variables. (2) The subtree that is swapped into the program tree may also contain dangling variables that were previously supported by declarations that were not part of the genetic material transferred. Both of these situations are resolved with the same repair operation, described in Section IV-D. An alternative solution could be to prevent crossovers

TABLE I: Type restrictions for each node's children. d is used to indicate that any pre-specified data-type is applicable. A `Void` data-type indicates no value is returned.

Node	Child data-types	Child node-types
`ForLoopDecl`	`Integer` `Void`	`Expression` `CodeBlock`
`ForEachLoopDecl`	array of d `Void`	`Expression` `CodeBlock`
`Declaration`	d	`Expression`

that lead to dangling variables, but it seems unlikely that the algorithm will be able to take advantage of declarations if they are prevented from being exchanged.

D. Repair Operation

To remove all dangling variables introduced by the crossover and mutation operators, a repair operation is applied to each program after undergoing one of these genetic operators. The repair operation replaces any dangling variables with an in-scope variable of a compatible data-type. To do this, the program tree is traversed, with each node checked to see if it is a dangling variable. A node is defined as a dangling variable if it has a `Variable` node-type and if that variable does not exist in the updated syntax at that point. To ensure the syntax includes only in-scope variables, all syntax updates are applied as the tree is traversed. If a dangling variable is identified, then it is replaced with a variable selected at random from those in the updated syntax with the correct data-type. If there are no suitable alternative variables then the genetic operator must be discarded and reattempted.

E. Syntax

The dynamic syntax feature makes it possible to introduce new types of node which can perform our desired variable declarations. These nodes are considered supplementary to the list provided in [2], and will be used in conjunction with the node-types described there. Three node-types from that paper that are of particular relevance here are `CodeBlock`, `Statement` and `Expression`. A `CodeBlock` contains a specified number of `Statement` nodes which get evaluated in sequence. We arbitrarily chose to use code-blocks of size 3. `Statement` and `Expression` are both abstract node-types, which have concrete subtypes such as `Assignment/IfStatement/ForLoop` and `Add/Variable/Literal` respectively. The following new node-types are all subtypes of `Statement` and have a `Void` data-type, which indicates they do not return a value. The type information for these nodes is listed in Table I.

- `Declaration` - Adds a variable to the syntax on its last syntax update. The value of this variable is set as the result of evaluating the child expression.
- `ForLoopDecl` - Adds an integer variable to the syntax on its second syntax update and removes the same variable on its third syntax update. On evaluation, the first child is evaluated to give the number of iterations (capped at 100). The child code-block is evaluated once per iteration, with the value of the variable set as the current index, starting from 1.
- `ForEachLoopDecl` - Adds a variable of the same data-type as its array input to the syntax on its second syntax update. The variable is removed on the third syntax update. On evaluation, the first child is evaluated to provide an array. The child code-block is evaluated once per element in the array, with the value of the variable set as the current element.

To limit the scope of variables to the code-block in which they are declared, the `CodeBlock` node records the state of the syntax on its first syntax update and reverts the syntax to that state on its final syntax update. This results in all variables declared within that block being removed from the syntax.

V. Experiments

A series of experimental runs were performed to test the impact of the introduction of variable declarations. Three different scenarios are compared, using different terminal and non-terminals sets. The labels SFGP, LOOP and DECL are used to distinguish the different experimental setups.

- SFGP - The terminal and non-terminal sets were made up of a problem specific selection of general purpose operators, without variable declarations. For two of the three problems, an identical setup is used as in [2] (the reverse-list problem was not used there).
- LOOP - As for SFGP, but each form of loop node was replaced with the equivalent declarative form. For instance, `ForLoop` was replaced with `ForLoopDecl`, which operates according to the same semantics, except that it declares its own variable for storing the iteration index.
- DECL - As for LOOP, but with the addition of a `Declaration` operator.

For each experimental setup, a sample of 500 runs were performed on three problems: factorial, even-n-parity and list reversion. These problems were chosen as they require programming constructs such as iteration and arrays to solve generally, and have been tackled previously in the literature [5], [14], [15]. A population of 500 and a maximum of 50 generations were allowed in all cases, with the crossover and mutation operators chosen from with probabilities 0.9 and 0.1 respectively. Tournament selection was used with a tournament size of 7. All other control parameters are outlined in tables II, III and IV. The experiments were conducted using the EpochX evolutionary framework [16], with operators implemented as specified in Sections III and IV.

On each problem, auxiliary variables are required for the normal, non-declarative loops and for returning a value. These variables are of an integer data-type for the factorial and list-reverse problems, and boolean for the even-n-parity problem. The initial value for the integer variables is 0, and is `true` for the boolean variables. Any auxiliary variables required for the SFGP setup are also supplied for the LOOP and DECL setups, even where not required. This is in order to keep the setups constant, other than the constructs under examination.

TABLE II: Parameter tableau for the factorial problem. `ForLoop` is used on SFGP, but replaced with `ForLoopDecl` for LOOP and DECL. `Declaration` is only used in the DECL setup.

Root data-type:	`Integer`
Root node-type:	`SubRoutine`
Max. depth:	6
Non-terminals:	`SubRoutine`, `CodeBlock`, `ForLoop/ForLoopDecl`, `Declaration`, `Assignment`, `Add`, `Subtract`, `Multiply`
Terminals:	i, $loopVar$, 1

A. Factorial

The task to be solved 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.

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

B. Even-N-Parity

The boolean parity problems are widely used as a benchmark task in the GP literature [1], [7]. However, they have only occasionally been tackled in the general form; for all values of `n` [14], [17]. A program which successfully solves the even-n-parity problem, must receive as input an array of booleans, arr, of unknown 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 [14]. The fitness of an individual is a simple count of how many of the 8 inputs are 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 solve the training cases.

C. Reverse List

A solution to the list reversion problem receives as input a list of variable length, and reverts the order of all elements in the list, returning the resultant list. In our experiments in this paper we use a list of characters, but any element data-type could equally have been used. The same five randomly generated lists of lengths 9..10 elements are used as the training inputs (shown below), with a further 30 randomly constructed lists of lengths 10..20 used to test generalisation.

TABLE III: Parameter tableau for the even-n parity problem. `ForEachLoop` is used on SFGP, but replaced with `ForEachLoopDecl` for LOOP and DECL. `Declaration` is only used in the DECL setup.

Root data-type:	`Boolean`
Root node-type:	`SubRoutine`
Max. depth:	8
Non-terminals:	`SubRoutine`, `CodeBlock`, `ForEachLoop/ForEachLoopDecl`, `Declaration`, `IfStatement`, `Assignment`, `And`, `Or`, `Not`
Terminals:	arr, $resultVar$, $loopVar1$, $loopVar2$, `true`, `false`

TABLE IV: Parameter tableau for the list reverse problem. `ForEachLoop` is used on SFGP, but replaced with `ForEachLoopDecl` for LOOP and DECL. `Declaration` is only used in the DECL setup.

Root data-type:	`Character[]`
Root node-type:	`SubRoutine`
Max. depth:	8
Non-terminals:	`SubRoutine`, `CodeBlock`, `ForLoop/ForLoopDecl`, `Declaration`, `ArrayLength`, `Subtract`, `Divide`, `Swap`
Terminals:	arr, $loopVar1$, $loopVar2$, 1, 2

The fitness of an individual is calculated as the sum of the levenshtein difference for each list, between the returned list and the correctly reverted list.

```
[U,V,B,L,N,U,G,D,A,H] [X,I,D,L,O,I,R,P,W]
[I,A,D,B,E,G,K,U,D] [C,R,T,U,U,U,P,W,N,M]
            [U,E,Q,W,G,U,O,M,O]
```

VI. Results & Discussion

The expectation was that the use of declarative loops would have a minimal impact on performance, but that the use of the `Declaration` node would severely degrade performance. This seemed likely because a `Declaration` which is introduced by a mutation operation will be unable to immediately contribute to the fitness of the individual. A second mutation is necessary to use the variable provided by that declaration in some productive way. Until that point, the declaration is effectively 'junk' code. So in our results, we would expect to see little difference between the SFGP and LOOP experimental setups, but for these both to display substantially better results than the DECL setup. The results summary in table V and related performance curves in Figures 2, 3 and 4, show that these expectations are only partially correct. The table displays the proportion of runs that found programs that were correct for all training and test cases (only training cases were used for fitness). It also shows the required computational effort, which is a calculation of the number of individuals that must be processed in order to produce a solution with 99% confidence [18]. The supplied confidence intervals for this value, are calculated using Wilson's 'score' method [19].

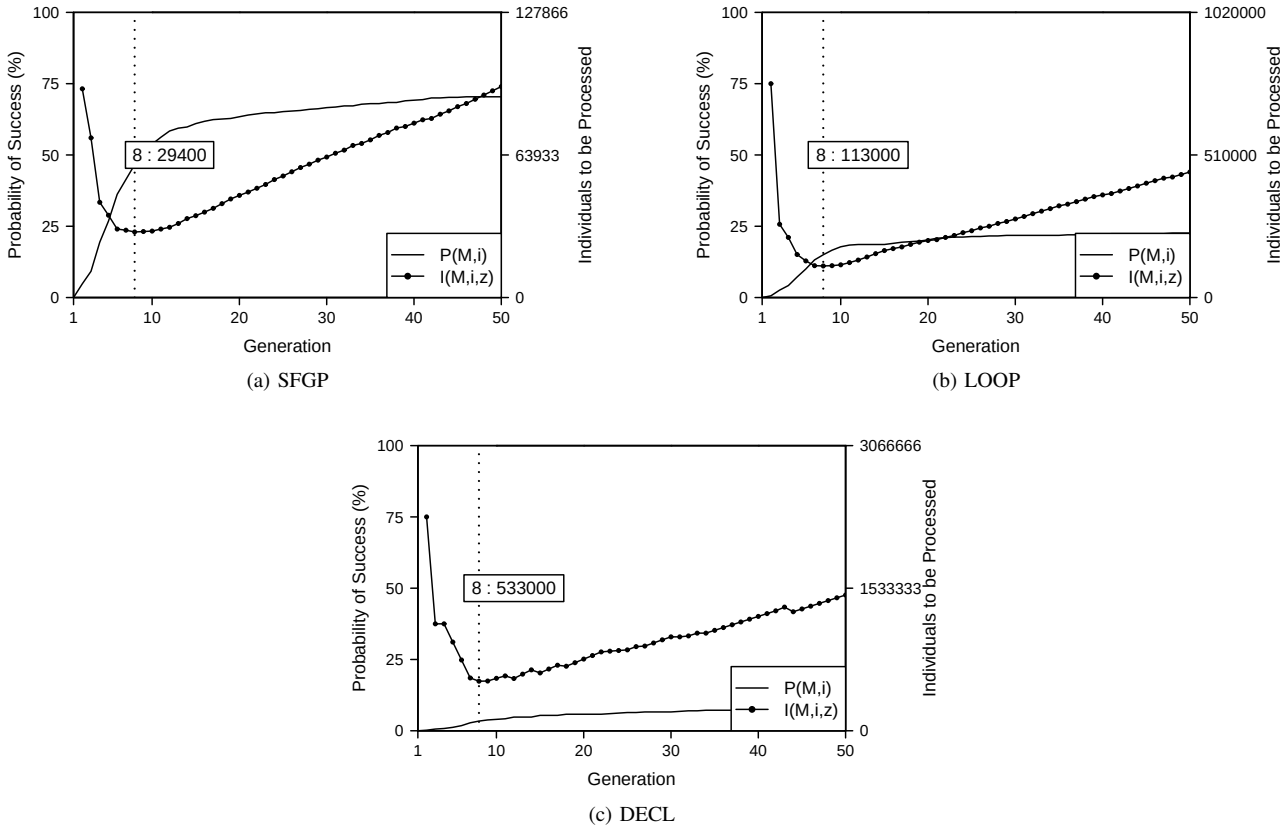


Fig. 2: Performance curves for the factorial problem. $P(M, i)$ is the success rate on the test cases and $I(M, i, z)$ is the number of individuals to process to find a solution with 99% confidence.

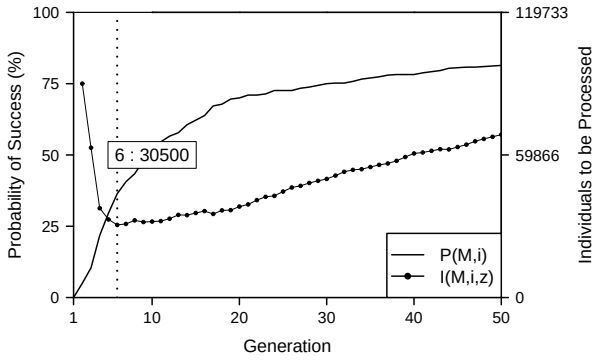
On the factorial problem, the DECL version does indeed perform terribly in comparison to SFGP. But, the results for LOOP are only slightly better. The results for the even-n-parity and list reverse problems are even more surprising. SFGP is the worst of the three setups on both of these problems. On the even-n-parity problem, LOOP shows a statistically significant drop in required computational effort compared to both DECL and SFGP. While on the list reverse problem, DECL has the lowest required computational effort, which is also a statistically significant result.

The results suggest the impact of variable declarations is problem dependent. It seems to be the case on the factorial problem, that there is an advantage in having index variables that will remain in scope beyond the last statement of a loop. It will be necessary to expand this experiment to a wider range of problems to get a better understanding of the circumstances in which variable declarations are beneficial to performance and when they are harmful. However, the primary motivations of this paper are not in the performance benefits. Allowing new variables to be declared is more consistent with how high-level imperative programming languages are used by human programmers. This is particularly true for the new loop constructs that supply their own limited scope variables. A second advantage, is that variable declarations can reduce the degree of insight required in to potential solutions. The need to consider the quantity and data-type of auxiliary variables to provide is removed if they can be successfully declared by the programs themselves.

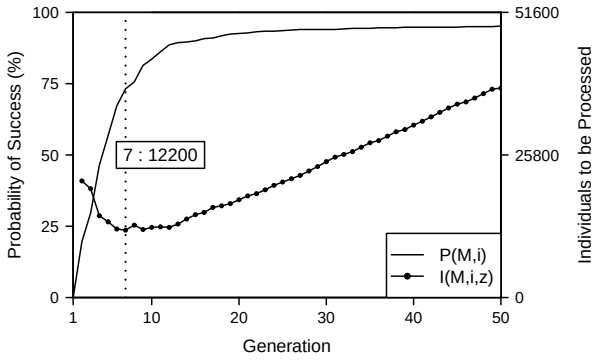
One concern with the modifications that were introduced, was that the need for a repair operation would be excessively damaging to the exchange of building blocks. The repair operation makes no consideration for the context of variables, and simply maps the dangling variables to new ones based on data-type. Table VI lists the proportion of individuals generated by each genetic operator that required one or more dangling variables to be repaired. Although we have not, as yet, examined the impact of the repair operation, it seems that any impact should be minimal since the proportion of individuals undergoing the operation is low. Note that mutations are unable to cause dangling variables with the syntax used for the LOOP experiments, because the only declarations available are provided by loops, where the scope of the variable is limited to one of its own subtrees. So if the declaration is replaced, so are all references to that variable.

VII. Conclusion

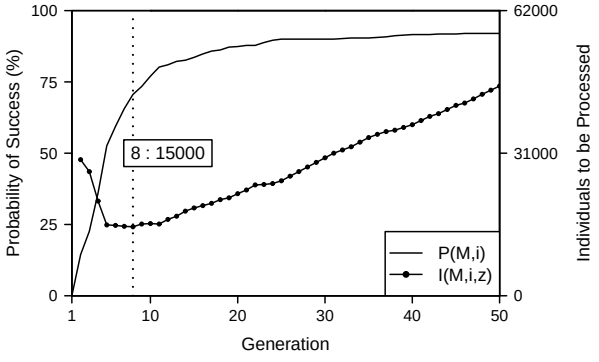
The addition of variable declarations can either improve or degrade evolutionary performance depending on the problem



(a) SFGP



(b) LOOP



(c) DECL

Fig. 3: Performance curves for the even-n-parity problem. $P(M, i)$ is the success rate on the test cases and $I(M, i, z)$ is the number of individuals to process to find a solution with 99% confidence.

TABLE V: Results summary, where *Exp* is the experiment, *Train%* is the probability of success on the training cases, *Test%* is the percentage of runs that found a solution that generalised to the test set. *Effort* is the required computational effort and *95% CI* is its confidence interval.

	Exp.	*Train%*	*Test%*	*Effort*	*95% CI*
Factorial	SFGP	70.8	70.6	29,400	25,800 - 33,500
	LOOP	22.8	22.8	113,000	90,600 - 142,000
	DECL	7.6	7.6	533,000	333,000 - 854,000
Parity	SFGP	91.6	81.4	30,500	26,400 - 35,400
	LOOP	99.0	95.2	12,200	11,000 - 13,700
	DECL	98.2	92.0	15,000	13,500 - 16,900
Reverse	SFGP	82.8	79.8	28,800	25,800 - 32,400
	LOOP	83.4	83.2	23,700	21,100 - 26,600
	DECL	88.4	87.8	20,000	17,900 - 22,400

and the way they are used. The `Declaration` nodes, that were solely for the purpose of adding new variables, were less damaging than expected, and in one case actually improved performance. However, the main advantage of declarations remains that they remove some of the challenge in allocating auxiliary variables, particularly where less is known about the solution in advance. The use of declarative loops was not entirely successful, and the substantial reduction in success rates on the factorial problem, in comparison to SFGP, demonstrates

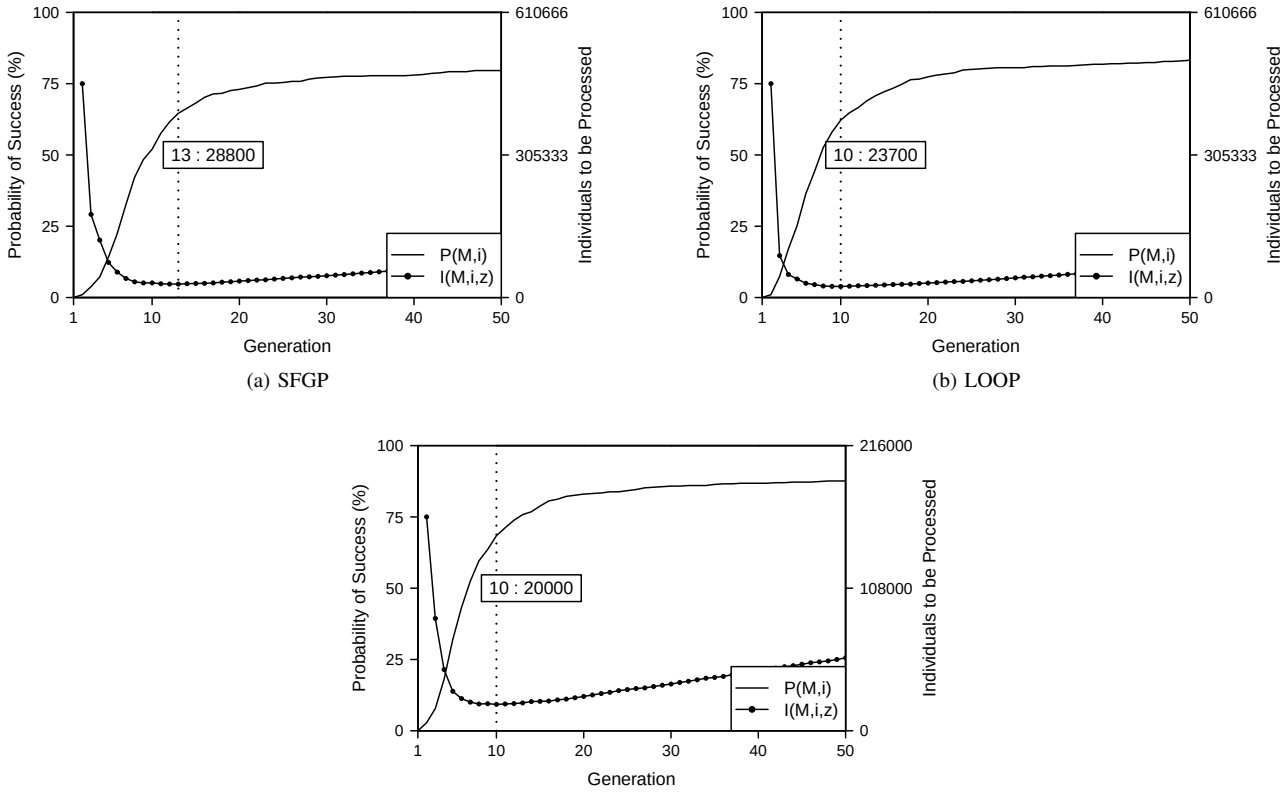
TABLE VI: Summary of repair operations, showing the percentage of program trees produced by each genetic operator that required the repair operation to fix one or more dangling variables.

	Exp.	Repair Operations	
		Crossover	*Mutation*
Factorial	LOOP	5.7%	-
	DECL	4.8%	0.6%
Parity	LOOP	16.3%	-
	DECL	16.3%	1.8%
Reverse	LOOP	12.1%	-
	DECL	13.3%	1.1%

that more work is needed to consider the circumstances in which they can be used productively. But use of loops that can declare their own variables will lead to programs that are more representative of human programming efforts. The value of loops generally is in producing smaller and more general solutions to more complex problems, so more work in this direction could be valuable.

REFERENCES

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



(a) SFGP

(b) LOOP

(c) DECL

Fig. 4: Performance curves for the list reverse problem. $P(M, i)$ is the success rate on the test cases and $I(M, i, z)$ is the number of individuals to process to find a solution with 99% confidence.

[2] T. Castle and C. G. Johnson, "Evolving high-level imperative program trees with strongly formed genetic programming," in *Proceedings of the 15th European Conference on Genetic Programming*, 2012.

[3] W. Wulf and M. Shaw, "Global variable considered harmful," *SIGPLAN Notices*, vol. 8, pp. 28–34, February 1973.

[4] V. Ciesielski and X. Li, "Experiments with explicit for-loops in genetic programming," in *Proceedings of the 2004 IEEE Congress on Evolutionary Computation*. Portland, Oregon: IEEE Press, 20-23 Jun. 2004, pp. 494–501.

[5] A. Agapitos and S. M. Lucas, "Learning recursive functions with object oriented genetic programming," in *Proceedings of the 9th European Conference on Genetic Programming*, ser. LNCS, vol. 3905. Budapest, Hungary: Springer, 2006, pp. 166–177.

[6] P. Nordin and W. Banzhaf, "Evolving Turing-complete programs for a register machine with self-modifying code," in *Genetic Algorithms: Proceedings of the Sixth International Conference*. Pittsburgh, PA, USA: Morgan Kaufmann, 1995, pp. 318–325.

[7] M. Brameier and W. Banzhaf, *Linear Genetic Programming*. Springer, 2007.

[8] M. Oltean and C. Grosan, "A comparison of several linear genetic programming techniques," *Complex Systems*, vol. 14, no. 4, 2004.

[9] T. Perkis, "Stack-based genetic programming," in *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, vol. 1. Orlando, Florida, USA: IEEE Press, 1994, pp. 148–153.

[10] L. Spector and A. Robinson, "Genetic programming and autoconstructive evolution with the push programming language," *Genetic Programming and Evolvable Machines*, vol. 3, no. 1, pp. 7–40, Mar. 2002.

[11] R. J. Abbott, "Object-oriented genetic programming, an initial implementation," in *Procceedings of the Sixth International Conference on Computational Intelligence and Natural Computing*, Cary, North Carolina USA, 2003.

[12] E. Kirshenbaum, "Genetic programming with statically scoped local variables," in *Proceedings of the Genetic and Evolutionary Computation Conference*. Las Vegas, Nevada, USA: Morgan Kaufmann, 2000, pp. 459–468.

[13] D. J. Montana, "Strongly typed genetic programming," *Evolutionary Computation*, vol. 3, no. 2, pp. 199–230, 1995.

[14] M. L. Wong and K. S. Leung, "Evolving recursive functions for the even-parity problem using genetic programming," in *Advances in Genetic Programming 2*. Cambridge, MA, USA: MIT Press, 1996, ch. 11, pp. 221–240.

[15] S. Harding, J. Miller, and W. Banzhaf, "Self modifying cartesian genetic programming: Fibonacci, squares, regression and summing," in *Proceedings of the 12th European Conference on Genetic Programming*, ser. LNCS, vol. 5481. Tübingen, Germany: Springer, 2009, pp. 133–144.

[16] T. Castle and L. Beadle, "Epochx: genetic programming software for research," http://www.epochx.org, 2007.

[17] L. Huelsbergen, "Finding general solutions to the parity problem by evolving machine-language representations," in *Genetic Programming 1998: Proceedings of the Third Annual Conference*. Madison, Wisconsin, USA: Morgan Kaufmann, 1998, pp. 158–166.

[18] D. Andre and J. R. Koza, "Parallel genetic programming on a network of transputers," in *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, J. P. Rosca, Ed., Tahoe City, California, USA, 9 Jul. 1995, pp. 111–120.

[19] M. Walker, H. Edwards, and C. Messom, "Confidence intervals for computational effort comparisons," in *Proceedings of the 10th European Conference on Genetic Programming*, ser. LNCS, vol. 4445. Valencia, Spain: Springer, 2007, pp. 23–32.