Preprint: To appear in: Genetic Programming Theory and Practice XIV (GPTP 2016). Springer.

Linear Genomes for Structured Programs

Thomas Helmuth, Lee Spector, Nicholas Freitag McPhee, and Saul Shanabrook

Abstract In most genetic programming systems, candidate solution programs themselves serve as genome upon which variation operators act. However, because of the hierarchical structure of computer programs and the syntactic constraints that they must obey, it is difficult to implement variation operators that affect different parts of programs with uniform probability. This lack of uniformity can have detrimental effects on evolutionary search, such as increases in code bloat. In prior work, structured programs were linearized prior to variation in order to facilitate uniform variation. However, this necessitated syntactic repair after variation, which reintroduced non-uniformities. In this chapter we describe a new approach that uses linear genomes that are translated into hierarchical programs for execution. We present the new approach in detail and show how it facilitates both uniform variation and the evolution of programs with meaningful structure.

Key words: uniform variation, linear genomes, Push, Plush

Thomas Helmuth
Computer Science, Washington and Lee University, Lexington, Virginia USA, e-mail: helmutht@wlu.edu

Lee Spector
Cognitive Science, Hampshire College, Amherst, MA USA, e-mail: lspector@hampshire.edu

Nicholas Freitag McPhee
Division of Science and Mathematics, University of Minnesota, Morris, MN USA, e-mail: mcphee@morris.umn.edu

Saul Shanabrook
Computer Science, University of Massachusetts, Amherst, MA USA, e-mail: s.shanabrook@gmail.com

1 Introduction

In traditional tree-based genetic programming, genetic operators such as subtree crossover and subtree mutation exhibit biases as to how likely it is for any given component of a parent program to be transferred to the resulting child. These biases make these genetic operators, and indeed any genetic operators defined over tree structures, decidedly nonuniform. What is meant by "uniform" in this context? In prior work [17] we defined uniformity in terms of two desiderata for variation operators:

- "that the probability of an inherited program component being modified during inheritance is independent of the size and shape of the parent programs beyond the component in question"
- "that pairs of parents are combined in ways that allow arbitrary combinations of components from each parent to appear in the child."

Genetic programming's most common program representation leverages the relative simplicity of Lisp symbolic expressions, which can express richly structured programs despite having few syntactic constraints in comparison to other common programming languages [4]. Hierarchical symbolic expressions, represented by tree structures, simplify the implementation of genetic operators that produce syntactically valid children from syntactically valid parents, using processes of subexpression replacement and exchange. However, the widely-used operators based on these processes do not meet our definitions for uniformity. For example, in standard subtree mutation a single subexpression is chosen and replaced, making the chance of replacing each subexpression inversely proportional to the size of the overall program. So standard mutation violates the first uniformity desideratum. In standard crossover a single subexpression is replaced by a subexpression from the other parent, restritcing the ways in which the components of the parents can be combined in children. Thus standard crossover violates the second uniformity desideratum.

Why do these deviations from uniformity matter? One issue is that the any particular subexpression is more likely to survive without modification if it is embedded within a large program rather than a small program. This can bias survival of important subexpressions toward larger programs, irrespective of fitness, leading to "code bloat" [9]. Another issue, related to the second uniformity desideratum, is that standard crossover does not permit complementary parts of two parents to be combined in their child, unless all of the needed parts from one parent are segregated in a single subexpression.

Several researchers have previously noted these and related issues and have attempted to address them through modification of the standard genetic operators. For example by making the probabilities of chosing a subexpressions dependent on it's size [4, 3], adjusting the number of replacements or exchanges [19], and by restricting exchanges to pairs of expressions with specified properties [12, 13, 14]. As detailed in [17], none of these methods meet our definition of uniformity, limited in principle by the nested structure of the programs that are being modified.

Genetic programming systems which use linear program representations are immune to most of the problems raised here, because uniform genetic operators can be straightforwardly applied on linear sequences [11]. Much of the prior work in linear genetic programming is focused on programs expressed in a low-level language with few control and data structures. Here we aim to provide uniform variation for programs in a language that can support arbitrary control and data structures and for which program structure is therefore likely to be more important. An existing framework in which linear genomes can indeed be used to evolve highly structured programs is "grammatical evolution," in which the genes on linear genomes are used as indices into grammars that can express arbitrary languages [15]. While uniform genetic operators can indeed be used on these genomes, the effects that small changes to genomes have on the expressed programs are often quite large, so that uniformity at the level of genomes is unlikely to translate into uniformity at the level of programs.

In earlier work, we sought to achieve greater uniformity by treating programs as linear sequences only during variation [17]. Our ULTRA (Uniform Linear Transformation with Repair and Alternation) operator first translates hierarchically structured programs into linear sequences, with parentheses replaced by independent tokens. It then applies uniform mutation and alternation (a form of multipoint crossover) to the linear sequences. Finally, it translates the resulting linear sequences back into hierarchical programs. Because the tokens for parentheses may have become imbalanced during uniform variation, a repair step is required to rebalance them. While the prior work demonstrated that ULTRA had several desirable properties, the artifacts produced by the repair step were themselves non-uniform and biased the shape of evolving programs in peculiar ways. This nonuniformity was one motivation for the work presented here.

Another motivation for the work described below was the fact that while ULTRA supports reasonably-uniform variation of structured programs, it does nothing to produce structure where it is most likely to be useful, in the context of instructions that make use of structure, such as conditional branches or loops.

The idea for the alternative approach that we present here arose when considering the problems raised above in the context of independent work that we were conducting in which "epigenetic" markers were added to instructions and literals in linear programs in order to turn those genes on or off [6, 7, 5]. We realized we could use similar epigenetic markers to specify the hierarchical structure for programs that are "expressed" from linear genomes. This allows us to perform uniform genetic operators on linear genomes and only express them as hierarchical programs for fitness testing. Furthermore, we specify that opening parentheses are automatically inserted following structure-dependent instructions during translation and use epigenetic markers to indicate where closing parentheses should occur. Thus we make it more likely that parenthesized, hierarchical structures appear in programs next to instructions that can make use of them.

We note that the use of the term "epigenetic" for these markers is most appropriate when they can change not only during reproduction, but also in their problem environments. While we do not describe such processes here, we have used these

markers in this way in the past [5]. In that work, we used hill climbing to modify the epigenetic markers of an individual if those modifications improve its fitness. While this effort did not produce significantly better results, using similar mutations to turn off or on genes in newly created children, and allowing selection pressure to sort out the changes, did produce impressively better results on 2 out of 5 problems. So, even though we do not explore changing epigenetic markers during the "lifetimes" of the programs in the present work, the markers that we use do enable such modifications; additionally, because of the ways that these markers are attached to instruction and literal "genes," we think that the use of the label "epigenetic" is reasonable in this context.

In the remainder of this chapter we first provide a brief description of Push the programming language, which is expressed from our linear genomes. We then describe our new linear genome representation, which we call "Plush" (where the "l" is for "linear"), in detail. This description is followed by experimental results that demonstrate the ways in which Plush facilitates program structure and the efficacy of various uniform genetic operators.

2 Push and PushGP

The Push programming language was developed specifically to serve as the target language for program evolution in genetic programming and related program synthesis methods (Spector, 2001; Spector and Robinson, 2002; Spector et al, 2005). Push is a postfix, stack-based language, which is similar in some respects to others that have been used for genetic programming (Perkis, 1994). When a Push program is executed, literals are pushed onto data stacks and instructions act on data that is on the stacks. Among the types of data stored on stacks and manipulated by instructions is `code`, which permits the expression of complex control structures via code manipulation. Program execution is implemented through the decomposition of programs and the processing of their instructions and literals on a special stack that contains code, the `exec` stack. Because all instructions take their arguments from appropriately typed stacks, and because of the Push convention that instructions finding insufficient data on the relevant stacks act as `no-op`s, instructions and literals of any types can be interleaved in arbitrary ways without risk of type errors.

Like Lisp symbolic expressions, Push programs may be hierarchically structured with parentheses, and this structure has consequences for program execution when code-manipulation instructions are used. Unlike symbolic expressions, parenthesized code blocks may appear anywhere in a Push program, and their presence or absence does not change the syntactic validity of the program. For example, the `exec_if` instruction will execute one of the top two items on the `exec` stack, depending on the value on top of the `boolean` stack, and discard the other. Those items serve as the conditional execution branches of the if statement, and either may be a single instruction or a code block containing any number of grouped instructions. For example, in the program:

```
(arg1 exec_if (4 5 integer_add) 7),
```

if `arg1` is *true*, `7` (as the "else" clause) will be removed, and the block of code `(4 5 integer_add)` will remain on the `exec` stack. If `arg1` is *false*, the "then" clause `(4 5 integer_add)` will be popped and `7` will remain on the `exec` stack.

In early versions of PushGP, the parenthetic structure of programs also affected the ways that genetic operators operated on programs. The genetic operators in these versions of PushGP were intentionally similar to those used in traditional, Lisp symbolic expression genetic programming systems; mutation involved replacing subexpressions with new subexpressions, while crossover involved the exchange of subexpressions across programs. This facilitated comparisons between the different program representations, and the translation of ideas from one project to another. But, these subtree-based operators lacked uniformity for the same reasons traditional tree-based operators do.

3 Plush

Plush[1] *genomes* provide an alternative representation for Push programs, storing programs in linear sequences that enable simple uniform genetic variations. There is a many to one mapping from Plush genomes to Push programs, as described by the translation process below.

One of the goals of introducing Plush genomes is to ensure that every argument taken from the `exec` stack for use by an instruction consists of a parenthesized block of code. In prior work, when evolving Push programs as genomes, we would often see `exec` stack manipulating instructions taking single instructions as arguments, instead of blocks of code. By ensuring that such instructions are followed by code blocks, we hope to encourage more modular programs that can make better use of `exec` stack arguments.

The many Push instructions that take arguments from the `exec` stack include instructions for looping, conditional execution, and other program manipulation. For example, the instruction `exec_if` requires two `exec` stack arguments, one to execute if the condition is true and the other to execute if the condition is false. The instruction `exec_do*times` needs one `exec` stack argument, which is executed repeatedly in a loop. In the program in Section 2, the `exec_if` instruction takes two arguments from the `exec` stack: `(4 5 integer_add)` and `7`. The first is a block of code, and the second is a single instruction (in this case, an integer literal). Note that blocks of code can contain zero or more instructions, so that the above program could be replaced with a functionally equivalent one where each of `exec_if`'s arguments is a block of code:

```
(arg1 exec_if (4 5 integer_add) (7))
```

[1] **L**inear **Push**

3.1 Structure

Plush genomes are linear sequences of *gene maps*, each of which contains, at minimum, an *instruction* and an epigenetic *close count* marker used to determine the placement of parentheses during translation. For example, below is a very simple Plush genome that encodes the Push program `(1 2 integer_add)`:

```
[ {:instruction 1, :close 0}
  {:instruction 2, :close 0}
  {:instruction integer_add, :close 0} ]
```

Plush gene maps can optionally contain other epigenetic markers. For example, the `:silent` marker contains a boolean that indicates whether the instruction should be included in the translated program. We have not yet added other epigenetic markers to Plush, but could imagine others being useful.

3.2 Translation

The process of translating a linear Plush genome into a syntactically valid, hierarchical Push program (which is a tree) is for the most part a depth-first construction of that program tree. The Plush genome is traversed linearly, adding each gene map's instruction to the end of the translated program. The hierarchical structure of the resulting program depends entirely on which of its instructions take arguments from the `exec` stack. Each instruction that does not take any arguments from the `exec` stack is simply appended to the growing program, and is not followed by a code block. An instruction that takes X arguments from the `exec` stack will be followed by X code blocks. After such instructions, further instructions will be added inside the opened code block, and will open nested code blocks when appropriate. Instructions only indicate where the code blocks open (i.e. they insert open parentheses); they do not describe where they should close (i.e. the location of the matching close parentheses).

As genes are translated from Plush into Push code, the values of the `:close` epigenetic markers determine where code blocks are closed. For every gene that is translated from Plush to Push: *after* the `:instruction` token has been added to growing Push program, and *after* a new block has been opened (if the instruction requires one), the `:close` marker is applied to the growing Push program. In particular, this number indicates the number of opened code blocks to close with closing parentheses. If the number is greater than the number of currently opened code blocks, all opened code blocks are closed. Note that if a code block is closed and the preceding instruction requires another code block (such as with `exec_if`, which requires two code blocks), one is immediately opened, which may be immediately closed if the `:close` marker is large enough. Finally, if the end of the genome is reached without closing all opened code blocks, the remaining blocks

are automatically closed, including any blocks that still needed to be opened for instructions that take multiple `exec` stack arguments.

These *automatic code blocks* ensure that the hierarchical structure of a program has semantic meaning according to its instructions. It may help to think about a language such as Python or Java, in which it makes sense to block off a chunk of code following the start of a loop, but does not make semantic (or syntactic) sense to have a block of code follow a variable assignment. While such semantically-irrelevant code blocks are syntactically valid in Push, they have no affect on the semantics of the program.

3.3 Special genes

The `:silent` epigenetic marker and two special instructions also affect the translation process:

- A Plush gene map with a `:silent` marker set to `true` is completely ignored and does not affect the growing Push program. Such genes have been "silenced."
- The `noop_open_paren` instruction immediately opens a new code block but adds no instruction to the Push program. No parenthesized branch is *ever* opened in the Push program unless the instruction takes one or more arguments from Push's `exec` stack or the instruction is `noop_open_paren`.
- The `noop_delete_prev_paren_pair` instruction restructures the Push program *without affecting the translation state in any other way*: it searches through the Push program until it finds the last block closed in translation, and "lifts" the contents of that block to the level of its parent in the program. For example, if the Push program is `[1 2 (3 4) 5 (6 *)]`, with the asterisk indicating where the next item would be added, inside a currently unclosed block, the result of applying this transformation would be `[1 2 3 4 5 (6 *)]`.

3.4 Example Translation

Here we give a brief example of a Plush genome and its corresponding Push program to illustrate the translation process. The genome:

```
[{:instruction exec_do*times :close 0}
 {:instruction 8 :close 0}
 {:instruction 11 :close 3}
 {:instruction integer_add :close 0 :silent true}
 {:instruction exec_if :close 1}
 {:instruction 17 :close 0}
 {:instruction noop_open_paren :close 0}
 {:instruction false :close 0}
```

Table 1 Genetic operator parameter settings used in all of our PushGP runs.

Parameter	Value
uniform mutation rate	0.01
constant tweak rate	0.5
uniform close mutation rate	0.1
close increment rate	0.2
alternation rate	0.01
alignment deviation	10

```
{:instruction code_quote :close 0}
{:instruction float_mult :close 2}
{:instruction exec_rot :close 0}
{:instruction 34.44 :close 0}))
```

is translated into the Push program:

```
(exec_do*times (8 11) exec_if
  ()
  (17
   (false code_quote (float_mult))
   exec_rot (34.44) () ()))
```

The first instruction, `exec_do*times`, takes one argument from the `exec` stack, and therefore opens one code block. The next two instructions are added to this block, which is closed by the 3 `:close` marker. Note that while this marker says to close 3 code blocks, there is only one open block to close. This block is followed by a silenced gene containing the instruction `integer_add`, which is not added to the Push program.

Next, the `exec_if` instruction takes two arguments from the `exec` stack. Since the `:close` count of the `exec_if` instruction itself is 1, the first of those two blocks is immediately closed, and the second opened. Following 17 in the opened block, the `noop_open_paren` instruction opens a code block without adding an instruction to the Push program.

In the remainder of the Plush genome, the `code_quote` instruction takes one `exec` stack argument, which is closed along with another code block after the instruction `float_mult`. Finally, `exec_rot` opens three code blocks, none of which are closed by the end of the program. As such, these blocks are automatically closed at the end of the program.

4 Uniform Genetic Operators

One of the advantages of a linear genome representation is it allows us to use uniform genetic operators. This section describes in detail the genetic operators we use

with Plush. For reference, Table 1 has the parameter settings related to genetic operators that we use in our experiments using the genetic operators described below, giving an idea of reasonable settings for these parameters; all indications thus far show these operators to be robust to changes in these parameter settings.

Uniform mutation modifies a single parent genome by changing each of its instructions with some probability, designated the *uniform mutation rate*. If an instruction in the genome is selected to be changed, we first check whether the instruction is a constant or a Push instruction. If it is an instruction, it is simply replaced by a random instruction from the instruction set. If it is a constant, there is a *constant tweak rate* probability of tweaking the constant; otherwise, it is replaced by a random instruction. The way in which a constant is tweaked depends on the type of the constant: integers and floats are perturbed by Gaussian noise with standard deviation of 1.0, strings have a 10% probability of replacing each character with a random character, and booleans are replaced by a random boolean.

While uniform mutation can change the instructions in a genome, it cannot affect the close epigenetic markers, and therefore cannot affect the structure of a program. We therefore created a *uniform close mutation* operator that takes a parent and alters its close markers. With *uniform close mutation rate* probability, it either increments or decrements the close marker associated with each instruction. The close marker cannot be decreased below 0, but has no upper bound. The probability of incrementing a close marker, as opposed to decrementing it, is given by the *close increment rate*; we typically keep this number less than 0.5, since otherwise we find that close markers tend to grow more than they shrink.

We use a crossover operator, *alternation*, heavily inspired by the ULTRA operator, which functioned on Push programs as genomes instead of linear genomes [17]. In alternation, both parents are traversed in parallel, copying instructions from one or the other into the child program. Before copying each gene, alternation has a small probability, the *alternation rate*, of moving the copying head to the other parent at the same index. Thus alternation copies sections of code from each parent into the child genome. In order to allow alternation to an index not identical to the prior index, we perturb the index with Gaussian noise, using the *alignment deviation* as the standard deviation for the perturbation. Thus the copy head may jump forward or backward during an alternation, but will not likely jump far.

Finally, we employ genetic operator pipelines to chain together two or more operators to create a single child. We mainly use this functionality to create a child genome by applying alternation and then uniform mutation.

The genetic operators described here meet the requirements of uniformity described in Section 1. In particular, all three operators give uniform probability of inheriting particular genetic material in a parent: with uniform mutation and uniform close mutation this probability is explicitly defined and with alternation it is roughly one half. Additionally, alternation allows “arbitrary combinations of components from each parent to appear in the child” [17], even though some of these combinations may be more likely than others based on position in the parent.

Table 2 Number of succesful runs out of 100. To be successful, a program has to perfectly pass all cases in the training set as well as an unseen test set. "Auto-Parens Off" is the version of the system where the locations of parentheses must be determined by evolution, instead of automatically.

Problem	Plush	Auto-Parens Off
Replace Space With Newline	51	51
Negative To Zero	45	34
X-Word Lines	8	0
Count Odds	8	5
Digits	7	9

5 Automatic Code Blocks Experiment

One of the primary motivations for developing Plush was to ensure that control instructions, which take arguments from the `exec` stack, have code blocks as arguments instead of single instructions. As discussed above, Plush automatically opens one or more parenthesized code block following each instruction that requires one or more arguments from the `exec` stack. Here, we conduct an experiment to examine the utility of automatic code blocks.

For this experiment we created a system that does not automatically create code blocks following specific instructions, but otherwise has similar characteristics to Plush. We started with Plush and removed the automatic opening of code blocks following specific instructions. We then added to the instruction set copies of `noop_open_paren`, as described in Section 3.3. Since we expect code blocks should be more common than other instructions, we added a number of copies of `noop_open_paren` to the instruction set to make a random program have a similar number of open parentheses as when using Plush with automatic parentheses; this resulted in around 30 copies added to about 150 other instructions, varying slightly per problem and instruction set. Otherwise, this method uses the same implementation as Plush, including close parenthesis markers. We call this method *Auto-Parens Off* for this experiment.

We compare Plush having automatic parentheses on and off using five general program synthesis benchmark problems. These problems require programs to manipulate multiple data types and use control flow structures. As such, we expect solutions to these problems will likely need to use hierarchical structure of code blocks in order to solve the problem, although solutions are possible without such structure. For more details about each problem, see their definitions in [2].

We conducted 100 runs with automatic parentheses on and 100 with them off on each problem. Table 2 gives the number of successful runs, i.e. runs that found a solution program that passed all of the training cases as well as every case in an unseen test set. The only problem that showed a significant difference between the systems is X-Word Lines, where native normal Plush was significantly better than with auto parenthesis off; in fact the set of runs with them off found no solutions at all.

We examined a sample of the solution programs from each problem. With the exception of the Replace Space With Newline problem, every solution program made semantic use of code blocks; in other words, each contained a code block that was an argument to an instruction that manipulated the `exec` stack. This was true both of programs using automatic parentheses and those that did not. On the other hand, almost all of the solutions to the Replace Space With Newline problem did not make semantic use of code blocks.

While these results do not make a strong case for the importance of automatic code blocks, they do hint at its power, specifically on the X-Word Lines problem. In this problem, a program must take an input string and an integer X, and should print the string with exactly X words on each line. Interestingly, every solution to this problem had at least two layers of nested, semantically-meaningful code blocks, which we did not see in many of the solutions to other problems. It may be the case that finding the correct position for one set of parentheses does not drastically hinder evolution without automatic parentheses, but correctly nesting multiple sets of parentheses significantly increases the difficulty.

6 Uniform Genetic Operators Experiment

As described in Section 4, the linear genomes of Plush allowed us to implement uniform genetic operators that don't exhibit the drawbacks often associated with tree-based genetic operators. Here we explore the efficacy of each of these operators by comparing sets of runs using different combinations and probabilities of operators. We tested five different treatments consisting of different probabilities of each genetic operator; these treatments are detailed in Table 3. Note that while no prior work has formally compared different settings of these operators, previous studies using Plush genomes [2, 1, 10] used the treatment REG.

We conducted trials with 100 GP runs using each treatment on five general program synthesis benchmark problems; again, see [2] for details of these problems. We present the results of these tests in Table 4.

While all treatments lead to relatively similar results, the treatment that performed most differently from the others is NUM (No Uniform Mutation). This treatment primarily uses alternation, with a small percentage of uniform close mutation operators. NUM had much lower success rates on all problems compared to REG, with a chi-squared significance test (with Holm correction) indicating significant differences at the 0.05 level on Replace Space With Newline, Negative To Zero, and X-Word Lines. This result indicates that uniform mutation has the largest role in determining success of PushGP on these problems.

Since runs without uniform mutation performed worst of our three treatments each leaving out a single operator, we decided to try a treatment only using uniform mutation, with results in the OUM column of Table 4; note that this treatment is very similar to NA in operator probabilities. While the success rates are lower than REG across the board, they are not significantly worse than REG on any problem.

Table 3 The probabilities of using each genetic operator to create a child for the five different treatments used in our genetic operator experiments. The operators, listed by their abreviations, are: "Alt." = alternation, "Uni. Mut." = uniform mutation, "Close Mut." = close mutation, and "Alt. + Uni. Mut." = alternation followed by uniform mutation.

Treatment	Description	Alt.	Uni. Mut.	Close Mut.	Alt. + Uni. Mut.
REG	Regular Operators	0.2	0.2	0.1	0.5
NCM	No Close Mut.	0.22	0.22	0	0.56
NUM	No Uni. Mut.	0.9	0	0.1	0
NA	No Alt.	0	0.9	0.1	0
OUM	Onlt Uni. Mut.	0	1.0	0	0

Table 4 Number of succesful runs out of 100 trials for different genetic operator treatments on five program synthesis problems; see Table 3 for treatment details. To be successful, a program has to perfectly pass all cases in the training set as well as an unseen test set.

Problem	REG	NCM	NUM	NA	OUM
Replace Space With Newline	51	50	24	55	41
Syllables	18	20	7	9	7
Negative To Zero	45	41	11	46	40
X-Word Lines	8	12	0	1	1
Count Odds	8	5	0	6	1

Even so, these results indicate that uniform mutation is not sufficient to produce the best results on its own, but works best in tandem with the other operators. Note that if a particular instruction disappears from the population, alternation and close mutation are not able to reintroduce it; we hypothesize that uniform mutation may provide the important ability to never get stuck in a population that cannot recover a useful lost instruction. Additionally, uniform mutation allows evolution to perform local search by changing small numbers of instructions, the importance of which has recently been noted [18].

The results do not indicate strongly whether uniform close mutation is particularly helpful or harmful. NCM, which does not use close mutation, gave results almost identical to REG. Another interesting comparison is NA, which uses 90% uniform mutation and 10% close mutation, and OUM, which uses 100% uniform mutation. While the differences between NA and OUM are not significant, NA does better on 4 of the 5 problems. Here, close mutation may be helping change the hierarchical shape of programs, which is possible through alternation but not by uniform mutation alone. Note that we never use close mutation more than 10% of the time, making it difficult to ascertain its importance.

Finally, even though these experiments show some differences between genetic operator treatments, those differences are overall minor and rarely statistically significant. These results show that the Plush representation is robust to major differences in genetic operator probabilities, as long as uniform mutation is included in some respect. This means practitioners need not worry about finding perfect settings

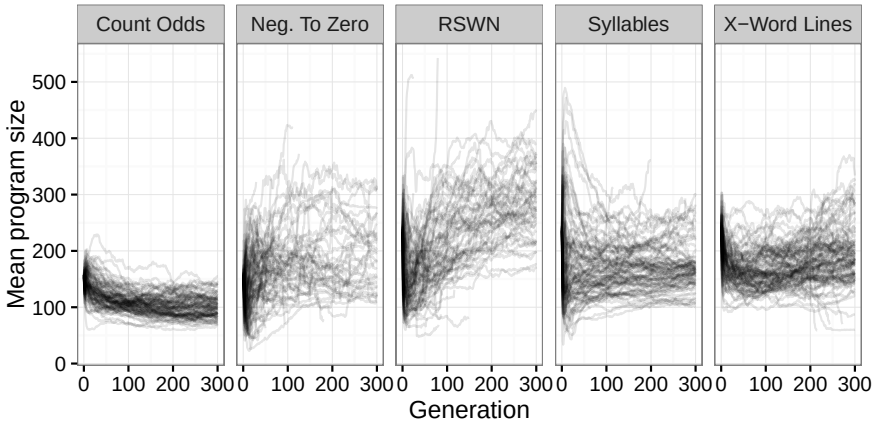

Fig. 1 Mean program sizes each generation for 100 runs each of five different software synthesis benchmark problems. Each run is plotted as a distinct line. "RSWN" is an abbreviation for "Replace Space With Newline".

for genetic operators, but instead can choose any reasonable settings and expect to not be worse than another setting.

6.1 Bloat

Code bloat without corresponding improvement in fitness has long caused problems in genetic programming [16, 8]. In our experience with uniform genetic operators in Plush, we have not observed code bloat. Figure 1, for example, plots the mean program size of each population over time for each run using the REG genetic operators. One of the problems (Count Odds) shows a slight *decrease* in average program size, and one (Replace Space With Newline) shows moderate growth. The mean program size for the other three problems remains fairly flat. In these runs, the maximum program size limit for the Count Odds and Negative to Zero problems is 1000, and for the other three problems is 1600. None of the mean program sizes come close to approaching these limits.

Of the uniform genetic operators we describe, only alternation can create a child of a different size than its parent. In fact, alternation has a slight bias toward creating smaller children than their parents, since alternation terminates upon reaching the end of the current parent's genome. This bias may partially account for the bloat control observed here, though children of alternation may also be larger than their parents. On the other hand, even operators that do not change the size of the produced children such as uniform mutation may have anti-bloat effects. For example, a bloated program will likely have more of its instructions replaced by uniform

mutation than a smaller program, increasing the chances of changes that break the functionality of the parent.

7 Conclusions

We have described a linear representation (Plush) for structured programs (in the Push programming language), and shown that it allows for uniform genetic operators that produce meaningful structure while solving difficult problems. The central idea of the representation scheme is to use epigenetic markers, attached to instructions and literals, to indicate where structure should be added to programs when they are expressed from the linear genomes. We compared the efficacy of different combinations of uniform genetic operators operating on Plush genomes and showed how the Plush-to-Push translation scheme encourages the expression of programs with structure in appropriate places. We note that the Plush-based system appears to be relatively robust to settings of the genetic operators and that it is capable of solving difficult software synthesis problems without producing significant code bloat.

Acknowledgements This material is based upon work supported by the National Science Foundation under Grants No. 1129139 and 1331283. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

1. Helmuth, T., McPhee, N.F., Spector, L.: Lexicase selection for program synthesis: a diversity analysis. In: Genetic Programming Theory and Practice XIII, Genetic and Evolutionary Computation. Springer (2015)
2. Helmuth, T., Spector, L.: General program synthesis benchmark suite. In: GECCO '15: Proceedings of the 2015 on Genetic and Evolutionary Computation Conference, pp. 1039–1046. ACM, Madrid, Spain (2015). DOI doi:10.1145/2739480.2754769. URL http://doi.acm.org/10.1145/2739480.2754769
3. Helmuth, T., Spector, L., Martin, B.: Size-based tournaments for node selection. In: M. Nicolau (ed.) GECCO 2011 Graduate students workshop, pp. 799–802. ACM, Dublin, Ireland (2011). DOI doi:10.1145/2001858.2002095
4. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge, MA, USA (1992). URL http://mitpress.mit.edu/books/genetic-programming
5. La Cava, W., Helmuth, T., Spector, L., Danai, K.: Genetic programming with epigenetic local search. In: GECCO '15: Proceedings of the 2015 on Genetic and Evolutionary Computation Conference, pp. 1055–1062. Madrid, Spain (2015)
6. La Cava, W., Spector, L.: Inheritable epigenetics in genetic programming. In: R. Riolo, W.P. Worzel, M. Kotanchek (eds.) Genetic Programming Theory and Practice XII, Genetic and Evolutionary Computation, pp. 37–51. Springer, Ann Arbor, USA (2014)
7. La Cava, W., Spector, L., Danai, K., Lackner, M.: Evolving differential equations with developmental linear genetic programming and epigenetic hill climbing. In: GECCO

Comp '14: Proceedings of the 2014 conference companion on Genetic and evolutionary computation companion, pp. 141–142. ACM, Vancouver, BC, Canada (2014). DOI doi:10.1145/2598394.2598491
8. Luke, S.: Issues in scaling genetic programming: Breeding strategies, tree generation, and code bloat. Ph.D. thesis, Department of Computer Science, University of Maryland, A. V. Williams Building, University of Maryland, College Park, MD 20742 USA (2000). URL http://www.cs.gmu.edu/ sean/papers/thesis2p.pdf
9. Luke, S., Panait, L.: A comparison of bloat control methods for genetic programming. Evolutionary Computation **14**(3), 309–344 (2006)
10. McPhee, N.F., Donatucci, D., Helmuth, T.: Using graph databases to explore the dynamics of genetic programming runs. In: Genetic Programming Theory and Practice XIII, Genetic and Evolutionary Computation. Springer (2015)
11. Oltean, M., Grosan, C., Diosan, L., Mihaila, C.: Genetic programming with linear representation: a survey. International Journal on Artificial Intelligence Tools **18**(2), 197–238 (2009). DOI doi:10.1142/S0218213009000111
12. Page, J., Poli, R., Langdon, W.B.: Smooth uniform crossover with smooth point mutation in genetic programming: A preliminary study. Tech. Rep. CSRP-98-20, University of Birmingham, School of Computer Science (1998). URL ftp://ftp.cs.bham.ac.uk/pub/tech-reports/1998/CSRP-98-20.ps.gz
13. Poli, R., Langdon, W.B.: On the search properties of different crossover operators in genetic programming. 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.) Genetic Programming 1998: Proceedings of the Third Annual Conference, pp. 293–301. Morgan Kaufmann, University of Wisconsin, Madison, Wisconsin, USA (1998). URL http://www.cs.essex.ac.uk/staff/poli/papers/PoliGP1998.pdf
14. 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). DOI doi:10.1023/A:1010068314282. URL http://citeseer.ist.psu.edu/335584.html
15. Ryan, C., Collins, J.J., O'Neill, M.: Grammatical evolution: Evolving programs for an arbitrary language. In: W. Banzhaf, R. Poli, M. Schoenauer, T.C. Fogarty (eds.) Proceedings of the First European Workshop on Genetic Programming, *LNCS*, vol. 1391, pp. 83–96. Springer-Verlag, Paris (1998). DOI doi:10.1007/BFb0055930. URL http://www.lania.mx/ ccoello/eurogp98.ps.gz
16. Silva, S., Costa, E.: Dynamic limits for bloat control in genetic programming and a review of past and current bloat theories. Genetic Programming and Evolvable Machines **10**(2), 141–179 (2009). DOI doi:10.1007/s10710-008-9075-9
17. Spector, L., Helmuth, T.: Uniform linear transformation with repair and alternation in genetic programming. In: R. Riolo, J.H. Moore, M. Kotanchek (eds.) Genetic Programming Theory and Practice XI, Genetic and Evolutionary Computation, chap. 8, pp. 137–153. Springer, Ann Arbor, USA (2013). DOI doi:10.1007/978-1-4939-0375-7_8
18. Trujillo, L., Z-Flores, E., Juárez-Smith, P., Legrand, P., Silva, S., Castelli, M., Vanneschi, L., Schütze, O., Muñoz, L.: Local search is underused in genetic programming. In: Genetic Programming Theory and Practice XIV, Genetic and Evolutionary Computation. Springer (2016)
19. Van Belle, T., Ackley, D.H.: Uniform subtree mutation. In: J.A. Foster, E. Lutton, J. Miller, C. Ryan, A.G.B. Tettamanzi (eds.) Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002, *LNCS*, vol. 2278, pp. 152–161. Springer-Verlag, Kinsale, Ireland (2002)