Semantic Forward Propagation for Symbolic Regression

Marcin Szubert[1(✉)], Anuradha Kodali[2,3], Sangram Ganguly[3,4], Kamalika Das[2,3], and Josh C. Bongard[1]

[1] University of Vermont, Burlington, VT 05405, USA
Marcin.Szubert@uvm.edu
[2] University of California, Santa Cruz, CA 95064, USA
[3] NASA Ames Research Center, Moffett Field, CA 94035, USA
[4] Bay Area Environmental Research Institute, Petaluma, CA 94952, USA

Abstract. In recent years, a number of methods have been proposed that attempt to improve the performance of genetic programming by exploiting information about program semantics. One of the most important developments in this area is *semantic backpropagation*. The key idea of this method is to decompose a program into two parts—a subprogram and a context—and calculate the *desired* semantics of the subprogram that would make the entire program correct, assuming that the context remains unchanged. In this paper we introduce Forward Propagation Mutation, a novel operator that relies on the opposite assumption—instead of preserving the context, it retains the subprogram and attempts to place it in the semantically right context. We empirically compare the performance of semantic backpropagation and forward propagation operators on a set of symbolic regression benchmarks. The experimental results demonstrate that semantic forward propagation produces smaller programs that achieve significantly higher generalization performance.

Keywords: Genetic programming · Program semantics · Semantic backpropagation · Problem decomposition · Symbolic regression

1 Introduction

Standard tree-based genetic programming (GP) searches the space of programs using traditional operators of subtree-swapping crossover and subtree-replacing mutation [4]. These operators are designed to be generic and produce syntactically correct offspring regardless of the problem domain. However, their actual effects on the behavior of the program, and thus its fitness, are generally hard to predict. For this reason, many alternative search operators have been recently proposed that take into account the influence of syntactic modifications on program semantics [1,10,11,13].

Semantic backpropagation [12,15] is arguably one of the most powerful techniques employed by such semantic-aware GP operators. The two operators based

J. Handl et al. (Eds.): PPSN XIV 2016, LNCS 9921, pp. 364–374, 2016.
DOI: 10.1007/978-3-319-45823-6_34

on semantic backpropagation—Random Desired Operator (RDO) and Approximately Geometric Crossover (AGX) have proved to be successful on a number of symbolic regression and boolean program synthesis problems [11,12]. Both operators rely on semantic decomposition of an existing program into two parts—a subprogram and its context. Given a subprogram, both operators attempt to calculate its *desired semantics*, i.e., the values that it should return to make the entire program produce the desired output, assuming that the context remains unchanged. The desired semantics can be then used to find a replacement for the subprogram that improves the overall program behavior.

Despite their superior performance when compared to other GP search operators [11,12,15], backpropagation-based RDO and AGX face a few major challenges that can limit their practical applicability. First of all, they are much more computationally expensive than traditional syntactic operators. Indeed, in order to calculate desired semantics, the target program output needs to be *backpropagated* by traversing the tree and inverting the execution of particular instructions. The computational cost of this operation is similar to the cost of a single fitness evaluation (which is typically the most expensive component of GP). Moreover, using desired semantics to find a subprogram replacement usually requires even more computational effort. Finally, the results reported so far demonstrate that RDO and AGX tend to produce relatively large programs that are difficult to interpret and may suffer from overfitting.

In this paper, we introduce Forward Propagation Mutation (FPM), a novel semantic-aware operator that also relies on program decomposition but works in the opposite manner to semantic backpropagation. Instead of preserving the context and replacing the subprogram, forward propagation retains the subprogram and attempts to place it in the semantically right context. In contrast to semantic backpropagation, the FPM operator does not require an additional tree traversal and thus it incurs less computational overhead. Moreover, the experimental results obtained on a set of univariate and bivariate symbolic regression problems demonstrate that it achieves competitive performance in terms of the training error while producing much smaller programs that usually perform significantly better on the unseen test cases.

2 Semantic Genetic Programming

In order to incorporate semantic-awareness into genetic programming, most of the recently proposed methods adopt a common definition of program semantics, known as *sampling semantics* [13], which is identified with the vector of outputs produced by a program for a sample of possible inputs. In supervised learning problems considered here, where n input-output pairs are given as a training set $T = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$, semantics of a program p is equal to vector $\mathbf{s}(p) = [p(\mathbf{x}_1), \ldots, p(\mathbf{x}_n)]$, where $p(\mathbf{x})$ is a result obtained by running program p on input $\mathbf{x}$. Consequently, each program p corresponds to a point in n-dimensional semantic space and a metric d can be adopted to measure semantic distance between two programs. Furthermore, fitness of a program p can be calculated as

a distance between its semantics $\mathbf{s}(p)$ and the target semantics $\mathbf{t} = [y_1, \ldots, y_n]$ defined by the training set, i.e., $f(p) = d(\mathbf{s}(p), \mathbf{t})$.

The information about program semantics and the structure of the semantic space endowed by a metric-based fitness function can be exploited in many ways to facilitate the search process carried out by GP. Apart from numerous semantic search operators [1,10,11,13], the knowledge about semantics can be used to maintain population diversity [3], to initialize the population [2] or to drive the selection process [7]. All such semantic-aware methods are collectively captured by the umbrella term of semantic genetic programming [14]. Recently, a paradigm of behavioral program synthesis [5] has been proposed, which extends semantic GP by using information not only about final program results but also about behavioral characteristics of program execution.

3 Semantic Backpropagation

One of the most important methods in semantic GP is semantic backpropagation [12]. The key concept behind this method is *program decomposition*: a program p is treated as a function (i.e., it is deterministic and has no side effects) that can be decomposed into two constituent functions (subprograms) p_1 and p_2 such that $p(\mathbf{x}) = p_2(p_1(\mathbf{x}), \mathbf{x})$. In particular, if a program is represented as a tree, such decomposition can be made at each node—the inner function p_1 is expressed by the subtree rooted at the given node, while the outer function p_2 corresponds to the rest of the tree (also termed *context* [9], see left part of Fig. 1).

Semantic backpropagation assumes that the desired program output $p^*(\mathbf{x})$ can be produced by retaining the outer function and replacing just the inner one by another subprogram p_s, i.e., $p^*(\mathbf{x}) = p_2(p_s(\mathbf{x}), \mathbf{x})$. Starting from the desired program output $p^*(\mathbf{x})$, the backpropagation algorithm heuristically inverts the program execution to calculate the desired semantics of the subprogram p_s, i.e., the values it should produce to make the entire program correct. This idea has been employed to design two operators, AGX and RDO, which differ with respect to what they use as the desired program output $p^*(\mathbf{x})$. In this study, we focus on RDO, a mutation operator that assumes that target semantics $\mathbf{t} = [y_1, \ldots, y_n]$ is given *a priori* and thus values $p^*(\mathbf{x}_i) = y_i$ can be used as an input for the backpropagation algorithm.

An example of a mutation performed by RDO is illustrated in Fig. 1 and proceeds as follows. First, a random mutation node is selected in the parent program (denoted as a circle with a double border in Fig. 1). The subtree p_1 rooted at this node is removed from the tree and the backpropagation algorithm is applied to calculate the desired semantics of the replacement p_s that would make the offspring program return desired values. The algorithm starts from the root of the tree, where desired semantics is given by $\mathbf{t}$, and follows the path to the removed subtree. For each node it calculates the desired semantics of its child by invoking the Invert function (a detailed description of this function and the RDO operator in general can be found in [12]).

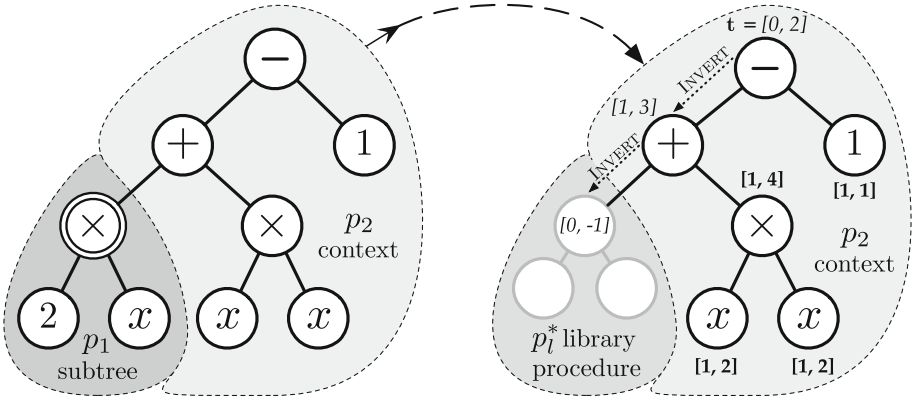

Fig. 1. A mutation performed by Random Desired Operator using semantic backpropagation. Desired semantics are denoted in italics.

For instance, let us assume that a training set contains just two cases with inputs $\mathbf{x} = [1, 2]$ and desired outputs $\mathbf{t} = [0, 2]$. As shown in Fig. 1, in the first step the algorithm finds out that to produce desired semantics at the root, knowing that outputs of its right child are equal to $[1, 1]$, the desired semantics of the left child must be equal to $[1, 3]$. This result is used in the subsequent step to calculate desired semantics for the next node. Finally, given desired semantics at the mutation node, the RDO operator attempts to replace the removed subtree with a subprogram that would produce such values. To this end, it employs a precomputed *library* of programs (procedures) that allows to efficiently retrieve a program p_l^* that has the smallest semantic distance to the desired semantics. Additionally, RDO also checks if a single constant real value would provide a better match to the desired semantics than p_l^*.

Importantly, in the process of semantic backpropagation, inverting certain functions can be ambiguous (if the function is not injective) or impossible (if the function is not surjective). As a result, the desired semantics may contain several values for each training case or special *inconsistent* elements. The library must be able to handle such queries efficiently [12,15].

4 Semantic Forward Propagation

Inspired by semantic backpropagation and RDO we propose an alternative mutation operator based on the complementary idea, which we term *semantic forward propagation*. Similarly to RDO, Forward Propagation Mutation (FPM) relies on decomposability of a program p into a subtree p_1 and a context p_2. However, while RDO assumes that a context can be preserved and attempts to replace the subtree, FPM makes the opposite assumption preserving the subtree and building a matching context for it.

The FPM operator starts by choosing a random mutation node in the parent program. The subtree p_1 rooted at this node is extracted from the tree and used

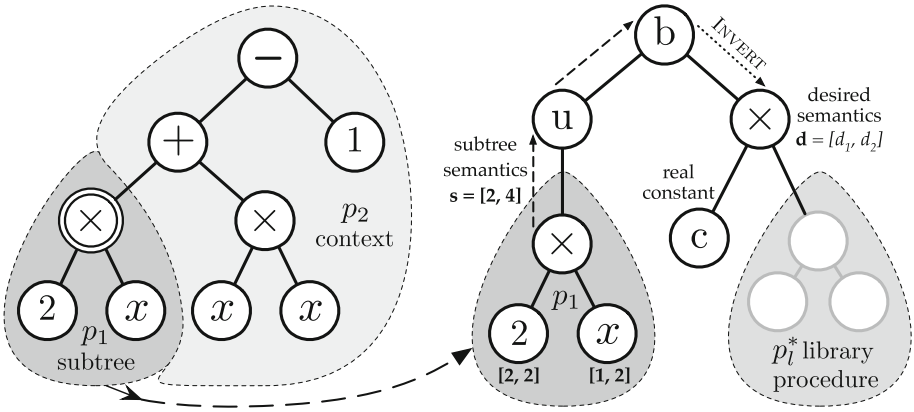

Fig. 2. An operation performed by Forward Propagation Mutation.

as a starting point for creating an offspring. In order to build a new context for this subtree, we assume a fixed structure of the context p_c containing 4 new nodes and a matching library procedure (see Fig. 2). We apply an exhaustive search to identify a context p_c^* of the assumed structure, that minimizes fitness of the entire offspring program $p_c^* = \arg\min_{p_c} f(p_c \circ p_1)$. To this end, we consider all pairwise combinations of the available unary (e.g., $\{\sin, \cos, \log, \exp\}$) and binary functions (e.g., $\{\times, +, -, /\}$) that could be placed directly above the selected subtree, as nodes u and b, respectively (cf. Fig. 2). Importantly, we extend the unary function set with the identity function $id(x) = x$. If the best found context p_c^* uses this function we skip adding the node u to the tree. For each pair of functions (u, b) placed above the subtree p_1, we *forward propagate* the semantics of the subtree up to the root of the new tree. Then, we apply just a single backpropagation step, using the same INVERT function as in RDO, to calculate desired semantics $\mathbf{d}$ of the other child of the node b, given the the target semantics $\mathbf{t}$ and the forward-propagated semantics $\mathbf{s}(u \circ p_1)$.

Since in this case the desired semantics is usually unambiguous, we can use a different method of searching the library, which could not be easily applied within the RDO operator. Here, we search for the library procedure which achieves highest cosine similarity. In other words, if we treat semantics as an n-dimensional vector, we return library procedure p_l^* that makes the smallest angle with the desired semantics $\mathbf{d}$, i.e.:

$$p_l^* = \arg\min_{p_l \in L} \arccos \frac{\mathbf{s}(p_l) \cdot \mathbf{d}}{\|\mathbf{s}(p_l)\| \|\mathbf{d}\|}.$$

Finally, we add a constant node c to scale the semantics of the library procedure making it closer to the desired semantics, i.e., $c = (\mathbf{s}(p_l^*) \cdot \mathbf{d}) \,/\, \|\mathbf{s}(p_l^*)\|^2$. An alternative, more computationally expensive approach, would be to run simple linear regression for each candidate program in the library, using its semantics as a single explanatory variable and desired semantics $\mathbf{d}$ as a response. This approach

would require extending the context structure to accommodate both an intercept and a slope coefficient.

5 Experimental Setup

The main goal of the experiments is to compare the performance of RDO and FPM mutation operators on a suite of symbolic regression benchmarks. Additionally, as a control setup we employ traditional subtree-replacing mutation (SRM). All three mutation operators are used along with conventional subtree-swapping crossover in a standard generational tree-based GP algorithm with tournament selection. Each mutation operator is employed in five setups with different values of mutation and crossover probabilities (the source code of our experiments is available at https://github.com/mszubert/ppsn_2016).

Most of the GP parameters (summarized in Table 1) are adopted from the recent work on semantic backpropagation [12]. In particular, whenever a random mutation/crossover node needs to be selected, a *uniform depth node selector* is used. Given a program p, it first calculates program's height h, then draws uniformly an integer d from the interval $[0, h]$ and finally selects a random node from all nodes at depth d in program p. This technique has been recently shown to reduce bloat when compared to conventional Koza-I node selectors [6,12].

Moreover, both RDO and FPM use population-based library which is constructed at each generation from all semantically unique subtrees (subprograms) in the current population. Since we impose an upper limit on the tree height (17), when searching the library we ignore all the procedures that would violate this constraint when inserted into the parent program.

We investigate training error, generalization performance (error on 1 000 unseen test cases) and the size of programs produced by using particular mutation operators on 11 symbolic regression benchmarks. We consider six univariate and five bivariate problems that are adopted from previous studies [4,8,12].

Table 1. Genetic programming parameters

Parameter	Value
Population size	256
Generations	100
Initialization	Ramped half-and-half with height range 2–6
	100 retries until accepting a syntactic duplicate
Instruction set	$\{+, -, \times, /, \exp, \log, \sin, \cos\}$ (log and/are protected)
Tournament size	7
Fitness function	Root-mean-square error (RMSE)
Node selection	Uniform depth node selector
Maximum tree height	17
Number of runs	30

Table 2. Symbolic regression benchmarks.

Benchmark name	Objective function	Variables	Training cases
P4 (Quartic)	$x^4 + x^3 + x^2 + x$	1	20
P7 (Septic)	$x^7 - 2x^6 + x^5 - x^4 + x^3 - 2x^2 + x$	1	20
P9 (Nonic)	$\sum_1^9 x^i$	1	20
R1	$(x+1)^3/(x^2 - x + 1)$	1	20
R2	$(x^5 - 3x^3 + 1)/(x^2 + 1)$	1	20
R3	$(x^6 + x^5)/(x^4 + x^3 + x^2 + x + 1)$	1	20
K11 (Keijzer-11)	$xy + \sin((x-1)(y-1))$	2	100
K12 (Keijzer-12)	$x^4 - x^3 + \frac{y^2}{2} - y$	2	100
K13 (Keijzer-13)	$6\sin(x)\cos(y)$	2	100
K14 (Keijzer-14)	$\frac{8}{2+x^2+y^2}$	2	100
K15 (Keijzer-15)	$\frac{x^3}{5} + \frac{y^3}{2} - x - y$	2	100

Selected benchmarks (see Table 2) include polynomial, rational and trigonometric functions. For each problem, fitness was calculated as root-mean-square error on a number of training cases. The univariate problems use 20 cases distributed equidistantly in the $[-1, 1]$ range, while the bivariate ones use a grid of $10 \times 10 = 100$ points spaced evenly in the $[-1, 1] \times [-1, 1]$ square.

6 Results and Discussion

Table 3 presents detailed characteristics of the best-of-run individuals evolved with particular mutation operators. Each row of the table corresponds to a single combination of one of the five GP setups (with different crossover (**X**) and mutation (**M**) probabilities) and one of the three considered mutation operators (either FPM, RDO or SRM). We performed 30 independent GP runs for each of such 15 combinations on each of the 11 symbolic regression problems. To confirm statistically significant differences between the results obtained with particular mutation operators, for each problem and parameters setup we conducted the Kruskal-Wallis test followed by a post-hoc analysis using pairwise Mann-Whitney tests (with sequential Bonferroni correction). We set the level of significance at $p \leq 0.05$. Table 3 shows with an underline the results that were found significantly better than those achieved with the other operators.

The first part of Table 3 shows the average training errors. Although RDO achieves the best overall results for most univariate problems, for the bivariate ones FPM produces more competitive results. Regardless of the parameter settings, the traditional SRM operator leads to the highest training error. Noteworthy, the RDO and FPM operators obtain their best results under different crossover and mutation settings. While both of them benefit from using traditional crossover as an additional variation operator, the performance of FPM

Table 3. Detailed characteristics of best-of-run individuals produced by particular mutation operators (FPM, RDO, SRM), aggregated over 30 GP runs. Each operator was employed in 5 GP setups with different crossover (**X**) and mutation (**M**) probabilites. Bold marks the best results achieved under certain **X**/**M** settings on particular problems. Underline indicates statistically significant superiority.

	X	M	P4	P7	P9	R1	R2	R3	K11	K12	K13	K14	K15
			Average training error										
FPM	0.0	1.0	**0.0011**	0.0072	0.0153	**0.0064**	0.0049	**0.0024**	**0.0626**	**0.0418**	**0.0000**	**0.0031**	**0.0061**
	0.5	0.5	**0.0001**	0.0018	**0.0025**	**0.0012**	0.0018	**0.0006**	**0.0299**	0.0111	**0.0000**	**0.0012**	**0.0007**
	0.5	1.0	**0.0001**	0.0025	0.0037	0.0020	0.0025	0.0007	0.0358	0.0154	**0.0000**	**0.0021**	**0.0007**
	1.0	0.5	**0.0000**	**0.0015**	0.0022	**0.0012**	**0.0013**	**0.0004**	**0.0283**	0.0086	**0.0000**	**0.0012**	**0.0004**
	1.0	1.0	**0.0001**	0.0026	0.0029	0.0018	0.0019	0.0006	0.0334	0.0116	**0.0000**	**0.0017**	**0.0007**
RDO	0.0	1.0	0.0030	**0.0034**	**0.0147**	0.0071	**0.0043**	0.0030	0.0709	0.0444	0.0440	0.0587	0.0302
	0.5	0.5	0.0004	**0.0017**	0.0029	0.0023	**0.0014**	0.0018	0.0455	**0.0090**	0.0038	0.0132	0.0024
	0.5	1.0	**0.0001**	**0.0008**	**0.0004**	**0.0006**	**0.0004**	**0.0002**	**0.0294**	**0.0029**	0.0007	0.0044	0.0015
	1.0	0.5	0.0003	0.0020	**0.0008**	0.0014	0.0015	**0.0004**	0.0504	**0.0063**	0.0015	0.0087	0.0041
	1.0	1.0	**0.0001**	**0.0003**	**0.0003**	**0.0008**	**0.0004**	**0.0004**	**0.0294**	**0.0047**	0.0011	0.0033	0.0008
SRM	0.0	1.0	0.0518	0.0742	0.0758	0.0744	0.0811	0.0097	0.2025	0.3049	0.1552	0.2145	0.0723
	0.5	0.5	0.0323	0.0968	0.0732	0.0834	0.0608	0.0156	0.1769	0.2328	0.1040	0.1138	0.0608
	0.5	1.0	0.0449	0.0926	0.0638	0.0792	0.0880	0.0115	0.1781	0.2128	0.1267	0.1603	0.0748
	1.0	0.5	0.0217	0.0882	0.0715	0.0663	0.0666	0.0078	0.1598	0.2005	0.0866	0.1690	0.0623
	1.0	1.0	0.0282	0.0845	0.0792	0.0698	0.0754	0.0120	0.1942	0.2479	0.1437	0.1724	0.0628
			Median test error										
FPM	0.0	1.0	**0.0009**	**0.0084**	**0.0342**	**0.0071**	**0.0044**	**0.0026**	**0.0555**	**0.0529**	**0.0000**	**0.0028**	**0.0057**
	0.5	0.5	**0.0000**	**0.0046**	**0.0256**	**0.0025**	**0.0123**	0.0030	**0.0425**	0.0581	**0.0000**	**0.0017**	**0.0008**
	0.5	1.0	**0.0000**	**0.0045**	**0.0142**	**0.0037**	**0.0045**	**0.0015**	**0.0333**	**0.0290**	**0.0000**	**0.0032**	**0.0008**
	1.0	0.5	**0.0000**	**0.0069**	0.0306	**0.0030**	**0.0042**	**0.0017**	**0.0260**	**0.0311**	**0.0000**	**0.0021**	**0.0005**
	1.0	1.0	**0.0000**	**0.0055**	**0.0227**	**0.0025**	**0.0027**	**0.0009**	**0.0300**	**0.0295**	**0.0000**	**0.0024**	**0.0008**
RDO	0.0	1.0	0.0039	0.0593	0.0346	0.0087	0.0145	0.0071	0.1089	0.0988	0.0215	0.0774	0.0185
	0.5	0.5	0.0025	0.5159	0.0469	0.0406	0.0148	**0.0028**	0.0738	**0.0374**	0.0070	0.0252	0.0036
	0.5	1.0	0.0117	0.3084	0.0715	0.1522	0.0652	0.0618	0.0639	0.2124	0.0022	0.0556	0.0097
	1.0	0.5	0.0006	0.0704	**0.0104**	0.0081	0.0319	0.0057	0.0445	0.0364	0.0030	0.0283	0.0014
	1.0	1.0	0.0097	19.486	8E+3	0.0607	0.0466	0.0155	0.0402	0.3878	0.0023	0.0378	0.0026
SRM	0.0	1.0	0.0485	0.1170	0.1017	0.0836	0.0585	0.0123	0.2005	0.2649	0.1986	0.1458	0.0814
	0.5	0.5	0.0240	0.0958	0.0810	0.0730	0.0592	0.0106	0.1770	0.1874	0.1122	0.0988	0.0525
	0.5	1.0	0.0572	0.1865	0.0922	0.0800	0.0694	0.0105	0.1686	0.2037	0.1311	0.1207	0.0882
	1.0	0.5	0.0191	0.0899	0.0785	0.0711	0.0641	0.0101	0.1493	0.1874	0.0998	0.1126	0.0446
	1.0	1.0	0.0257	0.0734	0.0725	0.0739	0.0727	0.0142	0.1894	0.1853	0.1843	0.1723	0.0389
			Average program size										
FPM	0.0	1.0	172.6	179.0	195.9	162.2	187.9	161.0	210.9	172.4	**9.1**	204.9	207.7
	0.5	0.5	150.4	341.4	322.1	325.3	352.8	347.7	328.3	305.5	**7.4**	326.5	260.6
	0.5	1.0	**78.3**	292.4	271.7	287.9	283.3	265.9	286.4	264.5	**8.5**	258.9	239.6
	1.0	0.5	**44.0**	346.6	354.4	327.2	339.2	311.0	328.0	311.1	**7.8**	298.6	300.0
	1.0	1.0	**99.2**	283.4	271.0	255.7	253.3	270.6	244.8	230.6	**8.9**	239.8	264.4
RDO	0.0	1.0	537.6	690.6	550.8	777.5	2656.9	1203.7	418.6	434.8	85.0	147.2	250.2
	0.5	0.5	503.6	637.9	686.0	493.9	529.6	485.7	358.4	482.4	497.1	346.4	1299.6
	0.5	1.0	626.9	1004.3	934.1	906.7	854.0	747.2	654.2	841.2	464.3	548.6	1137.2
	1.0	0.5	378.6	631.2	588.4	473.0	508.9	486.7	316.8	472.9	311.0	325.5	673.8
	1.0	1.0	645.6	903.6	909.9	668.6	746.4	696.2	542.9	838.7	426.7	514.6	1034.4
SRM	0.0	1.0	**122.9**	**176.1**	**152.4**	**133.8**	**116.2**	**155.9**	**109.3**	**95.1**	95.7	**63.0**	**74.7**
	0.5	0.5	**60.0**	**109.4**	**95.4**	**79.7**	**76.1**	**95.8**	**62.7**	**69.4**	59.6	**53.5**	**57.5**
	0.5	1.0	111.8	**172.8**	**159.6**	**154.3**	**122.3**	**173.6**	**99.2**	**95.3**	96.1	**79.1**	**82.0**
	1.0	0.5	97.9	**106.4**	**107.1**	**96.8**	**89.9**	**137.2**	**89.5**	**86.6**	81.1	**87.6**	**64.4**
	1.0	1.0	119.0	**160.9**	**147.5**	**150.6**	**131.0**	**165.7**	**95.3**	**83.2**	95.9	**80.4**	**96.5**

decreases when mutation is performed too frequently (i.e., if $\mathbf{M} = 1.0$). To explain this phenomenon let us note that for a given subprogram, the FPM operator builds a context in a deterministic way. As a result, if two semantically equivalent subprograms are selected in the same generation, they will result in identical offspring. Consequently, FPM can lead to creating too many duplicated programs and thus losing diversity in the population. Importantly, although RDO is also deterministic, it is less susceptible to this problem because typically the number of distinct contexts is much larger than that of distinct subtrees.

In order to assess generalization performance of evolved programs, we calculate the root-mean-square error on 1000 test cases drawn uniformly from the same range as for the training cases. The median test errors committed by the best-of-run individuals are presented in the second part of Table 3. In most cases, the RDO operator (especially for setups that achieve the lowest training error) suffers from substantial overfitting resulting in large test error. Although the FPM operator is also vulnerable to overfitting (in particular on problem P9) it is not as severe as in the case of RDO. With a few exceptions, for each of the considered problems and parameter setups, the FPM operator obtains the highest generalization performance.

Finally, we investigate the average size of best-of-run individuals which is presented in the last part of Table 3. Not surprisingly RDO is the most bloating operator and this is one of the reasons for its poor performance on the unseen test data. On the other hand, in preliminary experiments with imposed program size limit of 300 nodes, we also observed overfitting of the RDO operator. The programs produced by FPM tend to be much smaller. In particular, on two relatively simple problems, P4 and K13, the FPM operator finds short programs that obtain zero test error. Apparently, employing FPM allows to discover solutions that are very close to the original function underlying the training data. However, on all the other problems, the programs produced by RDO and FPM are significantly larger than those created by the traditional SRM operator.

7 Conclusions

Semantic GP operators have proved to be effective on a number of symbolic regression problems [11,13,14]. In this study, we confirmed these observations by analyzing the performance of the RDO operator based on semantic backpropagation [12] and the FPM operator that employs a novel idea of semantic forward propagation. When applied to a suite of symbolic regression benchmarks, both operators significantly outperformed the subtree-replacing mutation operator conventionally applied in GP. However, while both considered semantic operators achieved competitive performance on the training data, the RDO operator was found much more susceptible to overfitting. The proposed FPM operator, on the other hand, consistently produced shorter programs that obtained significantly lower error on the unseen test data.

Despite achieving superior predictive accuracy and producing shorter programs than RDO, the programs constructed with the FPM operator are still too

large to be easily understood. This is unfortunate since finding comprehensible solutions has been always considered as one of the primary benefits of using GP instead of black-box machine learning methods. As most semantic-aware operators tend to produce large or very large programs [10], the problem of bloat remains the major challenge that can limit the practical applicability of such methods. Therefore, one of the most important directions of future work is to investigate the performance of RDO and FPM operators combined with parsimony pressure mechanisms that control the complexity of evolved programs.

Acknowledgments. This work was supported by the National Aeronautics and Space Administration under grant number NNX15AH48G.

References

1. Beadle, L., Johnson, C.G.: Semantically driven crossover in genetic programming. In: Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2008, pp. 111–116. IEEE (2008)
2. Beadle, L., Johnson, C.G.: Semantic analysis of program initialisation in genetic programming. Genet. Prog. Evol. Mach. **10**(3), 307–337 (2009)
3. Jackson, D.: Promoting phenotypic diversity in genetic programming. In: Schaefer, R., Cotta, C., Kołodziej, J., Rudolph, G. (eds.) PPSN XI. LNCS, vol. 6239, pp. 472–481. Springer, Heidelberg (2010)
4. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
5. Krawiec, K.: Behavioral Program Synthesis with Genetic Programming, Studies in Computational Intelligence, vol. 618. Springer, Heidelberg (2016)
6. Krawiec, K., O'Reilly, U.M.: Behavioral programming: a broader and more detailed take on semantic GP. In: Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation, GECCO 2014, pp. 935–942. ACM (2014)
7. Liskowski, P., Krawiec, K., Helmuth, T., Spector, L.: Comparison of semantic-aware selection methods in genetic programming. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 1301–1307. ACM (2015)
8. McDermott, J., White, D.R., Luke, S., Manzoni, L., Castelli, M., Vanneschi, L., Jaskowski, W., Krawiec, K., Harper, R., De Jong, K., O'Reilly, U.M.: Genetic programming needs better benchmarks. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 791–798. ACM (2012)
9. McPhee, N.F., Hopper, N.J.: Analysis of genetic diversity through population history. In: Proceedings of the Genetic and Evolutionary Computation Conference, vol. 2, pp. 1112–1120. Morgan Kaufmann (1999)
10. Moraglio, A., Krawiec, K., Johnson, C.G.: Geometric semantic genetic programming. In: Coello, C.A.C., Cutello, V., Deb, K., Forrest, S., Nicosia, G., Pavone, M. (eds.) PPSN 2012, Part I. LNCS, vol. 7491, pp. 21–31. Springer, Heidelberg (2012)
11. Pawlak, T.P., Wieloch, B., Krawiec, K.: Review and comparative analysis of geometric semantic crossovers. Genet. Prog. Evol. Mach. **16**(3), 351–386 (2015)
12. Pawlak, T., Wieloch, B., Krawiec, K.: Semantic backpropagation for designing search operators in genetic programming. IEEE Trans. Evol. Comput. **19**(3), 326–340 (2015)

13. Uy, N.Q., Hoai, N.X., O'Neill, M., Mckay, R.I., Galván-López, E.: Semantically-based crossover in genetic programming: application to real-valued symbolic regression. Genet. Prog. Evol. Mach. **12**(2), 91–119 (2011)
14. Vanneschi, L., Castelli, M., Silva, S.: A survey of semantic methods in genetic programming. Genet. Prog. Evol. Mach. **15**(2), 195–214 (2014)
15. Wieloch, B., Krawiec, K.: Running programs backwards: instruction inversion for effective search in semantic spaces. In: Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation, GECCO 2013, pp. 1013–1020. ACM, New York (2013)