Improving Genetic Programming with Behavioral Consistency Measure

Krzysztof Krawiec[1] and Armando Solar-Lezama[2]

[1] Institute of Computing Science, Poznan University of Technology, Poznań, Poland
krawiec@cs.put.poznan.pl
[2] Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA
asolar@csail.mit.edu

Abstract. Program synthesis tasks usually specify only the desired output of a program and do not state any expectations about its internal behavior. The intermediate execution states reached by a running program can be nonetheless deemed as more or less preferred according to their information content with respect to the desired output. In this paper, a consistency measure is proposed that implements this observation. When used as an additional search objective in a typical genetic programming setting, this measure improves the success rate on a suite of 35 benchmarks in a statistically significant way.

Keywords: Program synthesis, genetic programming, entropy, multi-objective search.

1 Introduction

One of the main challenges for genetic programming (GP)—or for that matter, for any approach to program synthesis based on explicit search over a space of programs—is to decide on a fitness function that captures the relative quality of different proposed solutions. A common approach is to consider the quality of the *output* of a given program on a set of candidate inputs, possibly augmented with some structural constraints to prevent the search algorithm from overfitting to the training inputs (examples). The problem with this approach is that there is a fundamental mismatch between the search approach, which operates on the *structure* of the program, and the fitness function which is based on its input/output *behavior*. An ideal fitness function would instead be one that rewards programs that are structurally close to a correct solution: a program that is only a few small modifications away from being correct is better than one that has to be completely transformed in order to work, even if the former produces incorrect output on every input. The problem, of course, is that we do not know how the correct program looks like—if we did, we would not be searching for it—so we are left with behavior-based measures of correctness.

In this paper, we propose a new fitness measure that tries to better capture how close a proposed solution is from being correct by assessing the quality of intermediate values of the program in addition to the quality of the output.

T. Bartz-Beielstein et al. (Eds.): PPSN XIII 2014, LNCS 8672, pp. 434–443, 2014.

At first glance, it may seem that such a measure would run into the same problem as any other structural quality measure: since we do not know what the correct solution looks like, we also do not know what the intermediate values produced in the process of that computation should be. However, we do know that these intermediate values must preserve certain information about the input that is necessary to produce the desired output. Our main contribution is to show that information theoretic measures of the quality of intermediate values can improve upon traditional output-based measures in a statistically significant way.

The intuition for the approach is as follows: if two inputs are supposed to produce different outputs, then any program prefix that loses the information necessary to distinguish between these two inputs is doomed to failure since once lost, the information cannot be recovered. By contrast, if two inputs are supposed to produce the same output, then a program prefix that produces the same value for these two inputs will eliminate the risk that the rest of the program incorrectly assigns different values to these two inputs. The rest of the paper formalizes this intuition and exploits it in a new fitness measure based on *information consistency*. We implement the approach in a single-objective and a multi-objective variant, and evaluate it on 35 benchmarks representing three different domains. The experimental results clearly suggest that the involvement of information consistency almost always increases the likelihood of synthesizing the correct program.

2 Information Consistency

Conceptual Background. The proposed approach works by analyzing the internal behavior of programs, by which we mean the effects of computation conducted at intermediate stages of program execution. Conceptually, we examine such effects by placing *traps* (breakpoints) at selected locations in program code and inspecting the program state when execution reaches these traps. We assume that traps are not embraced by loops or conditional statements, thus ensuring that a deterministic program always visits all traps in the same order *independently of program input.*

When stopped at a trap, an executing program produces a certain *state* of the execution environment, which we assume to depend only on the part of the program that is syntactically associated with the trap (or, informally speaking, 'scoped' by the trap). Depending on the adopted programming paradigm, the particular interpretation of these notions will vary. For register-based sequential programs, a state would be the contents of all registers, and it would depend on the entire code executed so far (i.e., program prefix ending at the trap). For purely functional programming, by state we would mean the value calculated in the corresponding function call, caught in the process of being passed to the caller. In this study we consider functional tree-based GP and will place traps at tree nodes, so a state will be the value returned by the corresponding subtree.

Let s_i^k denote the state of the program at the kth trap when executed on the ith training example (input data). The sequence of the states $s_i^1, s_i^2, \ldots$ traversed

by a program for a given ith example will be called the *trace* for that example.

Our method is intended to promote programs that reach states that exhibit a certain form of consistency with the desired output. Given two examples for which the desired program outputs are y_i and y_j, two situations are possible:

Case #1: $y_i \neq y_j$. In this case, the program should maintain distinct behaviors for examples i and j. If that is not the case, i.e., the program leads to $s_i^k = s_j^k$ at some kth trap, this should be considered undesired and penalized. It is so because once program traces for two examples merge, they cannot diverge anymore (while they should if distinct y_i and y_j are to be produced at the end of program execution).

Case #2: $y_i = y_j$. In this case, the program should reach the same state for the ith and jth example at some stage of its execution, i.e., it is desirable to observe $s_i^k = s_j^k$. Conversely, a program for which no trap with this property exists is unlikely to end up with correct output, and thus should be penalized.

In other words, the observed execution states should form *equivalence classes* that are *consistent* with the equivalence classes induced by the desired program output. Ideally, a program would feature a trap for which this consistency is full, i.e., $s_i^k = s_j^k \iff y_i = y_j$. Once such a state emerges in an evolving program, producing the correct output is only a matter of one-to-one mapping from intermediate execution states to the final execution states. Although realizing such a mapping can be still difficult in some programming languages, we hypothesize that promoting programs that feature locations with such properties is desirable[1].

Consistency Measure. To promote and demote programs in an evolving population according to how their internal behaviors meet the characteristics discussed above, we design a measure based on information theory to quantitatively assess the consistency of program behavior with the desired output. Let us start from an observation that the process of program execution is usually accompanied by gradual loss of information in the execution environment. More precisely, a deterministic program can at most maintain the amount of information present in the data it has been applied to, but is unable to increase it. In an extreme case, a program that ignores its input and always returns the same output, reduces the amount of information to zero.

In terms of the notions introduced above, information is lost every time the traces associated with particular examples merge, i.e., every time the program, when applied to distinct training examples, reaches the same execution state (more specifically, if $s_i^k \neq s_j^k$ for a certain kth trap, but $s_i^l = s_j^l$ for some subsequent lth ($l > k$) trap). Depending on the particular pair of examples (i, j), that loss may be detrimental (when it increases the likelihood of producing the same output; case #1 in Section 2), or desirable (case #2).

[1] Whether the errors presented in Case #1 and #2 are *critical* depends on the adopted programming paradigm. For sequential programs, a state captures the entirety of computation conducted so far, so the erroneous merging or non-merging of traces cannot be fixed by subsequent execution of program suffix. For functional programming, however, other parts of the program can substitute for such a deficiency.

To assess the amount of lost information, we associate a random variable S_k with the kth trap, where the values of S_k are the states associated with particular examples. Analogously, we define a random variable Y representing the desired output. Based on the concept of conditional entropy ($H(X|Y) = -\sum \Pr(X|Y) \log_2 \Pr(X|Y)$), we consider:

- $H(Y|S_k)$, i.e., the amount of information that Y adds to S_k. In particular, if $H(Y|S_k) > 0$, then S_k alone is not sufficient to predict the value of Y.
- $H(S_k|Y)$, the amount of information that S_k adds to Y. Large values of $H(S_k|Y)$ indicate that S_k partitions the set of examples into many equivalence classes.

In connection to our previous considerations, every time the traces for two or more examples merge between the kth and $(k+1)$th trap, either the former term increases ($H(Y|S_k) > H(Y|S_{k+1})$) or the latter term drops ($H(S_k|Y) > H(S_{k+1}|Y)$). Both $H(Y|S_k)$ and $H(S_k|Y)$ attain zero if and only if S_k perfectly 'correlates' (is consistent) with Y, i.e., $s_i^k = s_j^k \iff y_i = y_j$.

Following this observation, we base our measure on the sum of the above terms. We define the (minimized) *information consistency* I of a program p according to the trap at which the total two-way conditional entropy attains its minimum, i.e.,

$$I(p) = \min_k H(Y|S_k(p)) + H(S_k(p)|Y) \tag{1}$$

where $S_k(p)$ is the random variable associated with the kth trap set on program p. The lower values of I are more desired as they indicate program behavior that is more consistent with Y. By using the minimum operator for aggregation over program traps, $I(p)$ rewards p for the part of its behavior that is most consistent with the desired output Y, even if otherwise (i.e., at other locations/traps) it behaves in a way that is unrelated to Y. This is intended to promote programs that are partially correct and thus feature code pieces that can prove useful in new programs[2].

Example. For simplicity, we illustrate these concepts on a linear program, by which we mean a program that reads in the input data only once, at the beginning of its execution and involves no loops or branches.

Consider a programming task defined by five examples, and a linear program with three traps. Table 1 presents the states traversed by the program for particular examples, where for brevity we encode program states in lowercase letters. We use different symbols for particular traps to emphasize that the random variables may assume values from different domains (though in practice, e.g., a, f, and j could represent the same value). The last column of the table presents the desired output of the program.

The table presents also the conditional entropy for consecutive traps. $H(Y|S_k)$ remains at zero for at the first and second trap (S_1 and S_2), and then increases

[2] Note that the term minimized in (1) can be alternatively expressed as $H(Y, S_k(p)) - I(Y; S_k(p))$, where $H(Y, S_k(p))$ stands for joint entropy and I is the mutual information. However, minimization of (1) is not equivalent to maximization of mutual information only, as $H(Y, S_k(p))$ may also vary from trap to trap.

Table 1. Exemplary calculation of consistency measure for a program with three traps, executed on five examples. The minimum of $H(Y|S_k)+H(S_k|Y)$ over the traps, marked in bold, is the information consistency I of this program.

Example	S_1	S_2	S_3	Y
1	*a*	*f*	*j*	*1*
2	*b*	*g*	*k*	*2*
3	*c*	*g*	*k*	*2*
4	*d*	*h*	*j*	*2*
5	*e*	*i*	*j*	*3*
$H(Y\|S_k)$	0	0	0.95	
$H(S_k\|Y)$	0.95	0.55	0.55	
$H(Y\|S_k) + H(S_k\|Y)$	0.95	**0.55**	1.50	

for S_3. As the states in consecutive traps merge (e.g., b and c in S_1 merge into g in S_2), the corresponding random variables S_k carry less and less information, and the entropy of Y conditioned on S_k grows.

Conversely, $H(S_k|Y)$ cannot grow with consecutive traps, because the collapsing states reduce the amount of information. In an ideal case, $H(Y|S_k) = H(S_k|Y) = 0$, i.e., neither the state adds any information to the desired output, nor the reverse. This would happen if g and h collapsed into a single state in S_2, which would then be perfectly correlated with Y. □

The motivations for our initial assumption that no trap is inside a loop or conditional statement should become clear now. Otherwise, a program could, depending on the input data, visit some traps more than once, not visit some traps, or visit them in a different order. In other words, the execution traces could not be 'aligned' in a reasonable way. This in turn would make it impossible to compare the corresponding execution states in a meaningful manner.

Related Work. In inspecting the internal behavior of programs, consistency measure relates to our former work on *behavioral search drivers* for GP [5]. However, the methods presented there were after a more general class of behaviors, and used machine learning inducers to identify them. Compared to them, here we focus exclusively on information contents, which allows us to assess the impact of this specific aspect of program behavior on search efficiency.

In a broader context, studying the internal behavior of programs can be seen as an extension of *semantic GP*, a new branch in GP research initiated by McPhee *et al.* [8]. However, most of work conducted in this area, including recent work (see, e.g., [9]), takes into account only the final output of programs.

The requirement that a program prefix should not loose information necessary to distinguish inputs that must lead to distinct outputs has been used in both constraint-based software synthesis [10] and interpolant-based hardware synthesis [3]. In different contexts, both of those works use the constraint to ensure that a given prefix can be completed to be equivalent to a given program.

Finally, the consistency measure proposed above will be integrated into evolutionary workflow, which can be done, among others, by treating it as an additional search objective. This can be considered as a form of *multiobjectivization* by Knowles *et al.* [4], meant as extending the original problem formulation by helper objectives intended to make problem solving more efficient.

3 Experiment

In the following, we examine the usefulness of our consistency measure in the framework of tree-based GP, with the programs being expression trees that feature no loops nor conditional statements, and have no side effects. A trap placed at a tree node will allow us to examine the value calculated by the program subtree (subprogram) rooted at that node. The state will be simply the value calculated by that subtree.

An important consequence of associating execution states with program subtrees is that a state does not capture the *entirety* of the computation conducted so far by a program, but only a *local* execution state (depending on the corresponding subtree; cf. Footnote 1). This however does not undermine the rationale presented in Section 2: a subtree that behaves consistently with the desired program outcome is a potentially useful piece of code, and a program that hosts it should be promoted. However, contrary to strictly linear programs, even if a subtree merges two or more states that should not be merged (and leads to $H(Y|S_k) > 0$), a program is not necessarily destined to perform bad, as other, independent program subtrees can still be able to distinguish those states.

Configurations. The goal of the experiment is to assess the impact on the information elicited by the consistency measure from the evolving programs. We consider two ways of integrating $I(p)$ (Formula (1)) into the conventional GP workflow: by redefining the scalar fitness in the conventional single-objective evolutionary workflow, and by using $I(p)$ as an additional objective in a multiobjective setting. In the former setup, called **FxI** in the following, the (minimized) program fitness is defined as

$$(1 + F(p))(1 + I(p)) \tag{2}$$

where $F(p)$ is the conventional program error (Hamming or city-block distance, depending on the domain). For the sake of simplicity, we deliberately abstain from exploiting the trade-off between F and I via weighing.

For the multiobjective setup (**FI** in the following) we assign a two-dimensional fitness $(F(p), I(p))$ to a program and employ the Non-Dominated Sorting Genetic Algorithm II (NSGA-II, [1]) at the selection stage, the arguably most popular method of multiobjective evolutionary optimization.

The baseline for the above methods is the conventional Koza-style GP (**F** in the following), which uses $F(p)$ as the only search objective.

To avoid making the (potentially biased) decisions where to set traps in programs, we stop program execution and calculate the two-way entropy (Eq. 1) after *every single instruction*. This imposes substantial computational burden on the evaluation process, which we take into account in one of the experiments.

A run is terminated when an ideal solution is found ($F = 0$) or the maximum number of 250 generations has elapsed. The percentage of runs that ended with the former result (out of the total of 30 independent evolutionary runs) forms the *success rate*, the performance metric we use in the following. A total of 30 $\times$ 35 benchmarks $\times$ 6 configurations = 6300 runs has been conducted.

Table 2. The benchmarks. v – number of input variables, m – number of tests, k – number of semantically unique programs.

Domain	Instruction set	Problem	v	m	k
Boolean	and, nand, or, nor	Cmp6, Maj6, Mux6, Par6	6	64	2^{64}
		Cmp8, Maj8, Par8	8	256	2^{256}
		Mux11	11	2048	2^{2048}
Categorical	$a_l(x, y)$	D-a1, D-a2, D-a3, D-a4, D-a5	3	27	3^{27}
	$a_l(x, y)$	M-a1, M-a2, M-a3, M-a4, M-a5	3	15	3^{15}
Regression	$+$, $-$, $*$, %, sin, cos, log, exp, $-x$	Keij1, Keij4, Nguy3..7, Sext	1		
		Keij5, Keij11..14, Nguy9..10, Nguy12	2	20	–
		Keij15	3		

For single-objective configurations, tournament of size 5 is used for selection. Except for the elements of the setup that have to differ across domains because of their different natures, all benchmarks in all considered domains use the same parameter settings. The remaining parameters use ECJ's defaults [6].

Program Synthesis Problems. Table 2 presents the 35 benchmark problems used in our experiment, which come from three domains: Boolean (8 benchmarks), categorical (10 benchmarks), and regression (17 benchmarks). Table 2 summarizes the problems, listing the instruction set, the number of variables, tests, and the cardinality of the search space (where countable). Note that none of the instruction sets contains constants.

The particular **Boolean problems** are defined as follows. For an v-bit comparator Cmp v, a program is required to return *true* if the $\frac{v}{2}$ least significant input bits encode a number that is smaller than the number represented by the $\frac{v}{2}$ most significant bits. In case of the majority Maj v problems, *true* should be returned if more that half of the input variables are *true*. For the multiplexer Mul v, the state of the addressed input should be returned (6-bit multiplexer uses two inputs to address the remaining four inputs, 11-bit multiplexer uses three inputs to address the remaining eight inputs). In the parity Par v problems, *true* should be returned if an only if an odd number of *true*s appears on the inputs.

The **categorical problems** come from Spector *et al.*'s work on evolving algebraic terms [11] and dwell in the ternary domain: the admissible values of program inputs and outputs are $\{0, 1, 2\}$. The peculiarity of these problems consists in using only one binary instruction in the programming language; for the a_1 algebra, the semantics of that instruction is defined in Table 3c. We refer the reader to [11] for the definitions of the remaining algebra problems.

For each of the five algebras considered here, we consider two tasks. In *discriminator term* tasks (*D*-* in Table 2), the goal is to synthesize an expression (using only the one given instruction) that accepts three inputs x, y, z and realizes the function given in Table 3a. Given three inputs and ternary domain, this gives rise to $3^3 = 27$ fitness cases for these benchmarks.

The second task defined for each of the algebras (*M*-* in Table 2), consists in evolving a so-called *Mal'cev term*, i.e., a ternary term that is equivalent to the one given in Table 3b. This condition specifies the desired program behavior only

Table 3. The algebra problems: (a) discriminator problem, (b) Mal'cev problem, (c) exemplary algebra (named a_1 in [11])

(a) $t(x,y,z) = \begin{cases} x & if\ x \neq y \\ z & if\ x = y \end{cases}$ (b) $m(x,x,y) = m(y,x,x) = y$ (c)

a_1	0	1	2
0	2	1	2
1	1	0	0
2	0	0	1

for the indicated combinations of inputs, and the desired value for all distinct inputs is not determined. As a result, there are only 15 fitness cases in our Mal'cev tasks, the lowest number of all considered benchmarks.

The **regression problems** considered here come from [7] and include both univariate and multivariate target functions. The univariate ones (*Keij1*, *Keij4*, *Nguy3..7* and *Sext*) use 20 tests uniformly distributed in the $[-1, 1]$ interval, except for the *Keij4* benchmark which uses the $[0, 10]$ interval. The remaining problems are predominantly bivariate, and involve $5 \times 5 = 25$ fitness cases uniformly distributed on the two-dimensional grid. The only exception is *Keij5*, which hosts three input variables, with $4 \times 4 \times 4 = 64$ fitness cases distributed equidistantly in the cube. For other details on these benchmarks, see [7].

Results for Limited Number of Evaluations. In this experiment, the computational budget allocated to every run is $250,000$ evaluations, i.e., a population of $1,000$ programs evolves for 250 generations. Because of the large number of benchmarks and limited space, we discuss only the aggregated outcomes. To this aim, we rank the three considered configurations according to the success rate on every benchmark independently. Next, we average the ranks across benchmarks.

When averaged over **all benchmarks**, the resulting ranks are FI: 1.6, F: 2.19, and FxI: 2.21 (the lower rank, the better). To assess the statistical significance of this outcome, we used the Friedman's test for multiple achievements of multiple subjects which, compared to ANOVA, does not require the distributions of variables in question to be normal. The p-value of 0.00124 strongly indicated that at least one method performed significantly different from the remaining ones. A post-hoc analysis using symmetry test [2] determined that the difference between F and FxI is statistically insignificant, but FI significantly outranks them both ($p = 0.03$).

We can conclude thus that augmenting the conventional training signal (program error) with our information-based behavioral measure that depends on internals of program execution (information consistency) brings substantial benefits to a GP search algorithm. However, this seems to hold only when the behavioral information is provided as a separate search objective (FI setup). Aggregation of conventional fitness with information consistency (FxI setup), at least in the specific multiplicative manner used here (Eq. 2) does not make the search more efficient. Possible explanation is that F, defined as city-block distance for regression, and normalized Hamming distance for the remaining two domains, may assume radically different ranges on different problems. As a result, the trade-off between F and I varies across domains and can be difficult to control.

FI fared the best on the **categorical problems**, where it came on top on nine out of ten benchmarks, so that its average rank was 1.2 there. Remarkably, this is also the only domain in which FxI ranked on average better than F (2.05 vs. 2.75). This may suggest that the trade-off between F and I was just right for these problems. For **regression problems**, the differences between all three methods were the least prominent (FI: 1.82, F: 2.0, FxI: 2.18), which was expected, as discrete entropy cannot capture similarities between values for these continuous problems.

On the **Boolean domain**, the behavioral methods performed relatively bad (FI: 1.62, F: 1.88, FxI: 2.5), which seems surprising given the discrete nature of these problems. We come up with three mutually nonexclusive hypotheses to account for this. Firstly, note that the Boolean instruction set does not feature negation (Table *2*). Let us consider a program p that evolves a subexpression p' which produces exactly the negated value of desired output Y. In terms of I, p' is perfectly consistent with Y, so $I(p) = 0$. However, without negation in the instruction set, it may be difficult to extend p' with a suffix that would turn it into Y. A way to achieve this is $p'\, nor\, false$ or $p'\, nand\, true$, but those Boolean constants require a separate subprogram to synthesize them.

The other hypothesis starts with the observation that Boolean problems feature the largest number of input variables in our benchmark suite ($v \geq 6$, while $v \leq 3$ for the other domains, Table 2). As no input variable is redundant in these problems, evolution has to produce a subprogram comprising at least v tree leaves (terminals), and thus $2v - 1$ nodes, to possibly bring I to zero. In general, to obtain competitive values of I, relatively large subtrees featuring most input variables have to be synthesized.

Last but not least, the random variables that I is based on, by being binary for the Boolean problems, are least discriminating in terms of entropy. As an example, a binary random variable observed for five training examples can have only one of three possible values of entropy, while an analogous number for a ternary variable (like those used in our categorical problems) is five. As a result, I can be more fine-grained for domains that feature multi-valued variables, even if the actual number of examples is low (like in our categorical problems).

Results for Limited Time Budget. To take into account the overhead of calculating consistency measure, in this experiment the computational budget allocated to every run was 600 seconds. For this setup, the average ranks w.r.t. success rate are: FI: 1.77, F: 2.06, FxI: 2.17. Though FI leads again, this time Friedman test is inconclusive at 0.05 level. However, its relatively low p-value (0.08) suggests that significance could be attained given a larger suite of benchmarks.

4 Summary

We proposed a measure for characterizing the internal program behavior in terms of its consistency with the desired output, which can conveniently be used to promote certain program behaviors without specifying them explicitly. The algorithms that involve this measure can be thus said to implicitly perform *problem*

decomposition, which normally requires an expert who explicitly splits a task into subtasks. Together with methods reported elsewhere [5], we consider information consistency as promising way towards scalable program synthesis.

Acknowledgment. K. Krawiec acknowledges support from the NCN grant no. DEC-2011/01/B/ST6/07318, and A. Solar-Lezama from grant no. NSF-CCF-1161775.

References

1. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Transactions on Evolutionary Computation 6(2), 182–197 (2002)
2. Hollander, M., Wolfe, D.: Nonparametric Statistical Methods. A Wiley-Interscience Publication. Wiley (1999)
3. Jiang, J.-H.R., Lee, C.-C., Mishchenko, A., Huang, C.-Y.R.: To sat or not to sat: Scalable exploration of functional dependency. IEEE Transactions on Computers 59(4), 457–467 (2010)
4. Knowles, J.D., Watson, R.A., Corne, D.W.: Reducing local optima in single-objective problems by multi-objectivization. In: Zitzler, E., Deb, K., Thiele, L., Coello Coello, C.A., Corne, D.W. (eds.) EMO 2001. LNCS, vol. 1993, pp. 269–283. Springer, Heidelberg (2001)
5. Krawiec, K., Swan, J.: Pattern-guided genetic programming. In: Blum, C., et al. (eds.) GECCO 2013: Proceeding of the Fifteenth Annual Conference on Genetic and Evolutionary Computation Conference, Amsterdam, The Netherlands, July 6-10, pp. 949–956. ACM (2013)
6. Luke, S.: ECJ evolutionary computation system (2002), `http://cs.gmu.edu/eclab/projects/ecj/`
7. 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: Soule, T., et al. (eds.) GECCO 2012: Proceedings of the Fourteenth International Conference on Genetic and Evolutionary Computation Conference, Pennsylvania, USA, July 7-11, pp. 791–798. ACM (2012)
8. McPhee, N.F., Ohs, B., Hutchison, T.: Semantic building blocks in genetic programming. In: O'Neill, M., Vanneschi, L., Gustafson, S., Esparcia Alcázar, A.I., De Falco, I., Della Cioppa, A., Tarantino, E. (eds.) EuroGP 2008. LNCS, vol. 4971, pp. 134–145. Springer, Heidelberg (2008)
9. 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)
10. Singh, R., Singh, R., Xu, Z., Krosnick, R., Solar-Lezama, A.: Modular synthesis of sketches using models. In: McMillan, K.L., Rival, X. (eds.) VMCAI 2014. LNCS, vol. 8318, pp. 395–414. Springer, Heidelberg (2014)
11. Spector, L., Clark, D.M., Lindsay, I., Barr, B., Klein, J.: Genetic programming for finite algebras. In: Keijzer, M., et al. (eds.) GECCO 2008: Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation, Atlanta, GA, USA, July 12-16, pp. 1291–1298. ACM (2008)