Geometric Semantic Genetic Programming for Recursive Boolean Programs

Alberto Moraglio
University of Exeter
Exeter, UK
A.Moraglio@exeter.ac.uk

Krzysztof Krawiec
Poznan University of Technology
Poznan, Poland
krawiec@cs.put.poznan.pl

ABSTRACT

Geometric Semantic Genetic Programming (GSGP) induces a unimodal fitness landscape for any problem that consists in finding a function fitting given input/output examples. Most of the work around GSGP to date has focused on real-world applications and on improving the originally proposed search operators, rather than on broadening its theoretical framework to new domains. We extend GSGP to recursive programs, a notoriously challenging domain with highly discontinuous fitness landscapes. We focus on programs that map variable-length Boolean lists to Boolean values, and design search operators that are provably efficient in the training phase and attain perfect generalization. Computational experiments complement the theory and demonstrate the superiority of the new operators to the conventional ones. This work provides new insights into the relations between program syntax and semantics, search operators and fitness landscapes, also for more general recursive domains.

CCS CONCEPTS

•Software and its engineering → Genetic programming; •Theory of computation → *Evolutionary algorithms;*

KEYWORDS

Genetic Programming, Semantics, Geometric Semantic Genetic Programming, Principled Design, Boolean Functions, Recursive Programs

ACM Reference format:
Alberto Moraglio and Krzysztof Krawiec. 2017. Geometric Semantic Genetic Programming for Recursive Boolean Programs. In *Proceedings of the Genetic and Evolutionary Computation Conference 2017, Berlin, Germany, July 15–19, 2017 (GECCO '17),* 8 pages.
DOI: 10.1145/3071178.3071266

1 INTRODUCTION

For about a decade, Genetic Programming (GP) witnessed a trend towards a deeper understanding of program semantics and its effect on search [2, 6, 7, 12], which often led to improved forms of GP. GSGP [8] is a form of semantic GP with a strong theoretical foundation which induces a simple unimodal fitness landscape for any supervised machine learning problem and has provably good search performance [9].

Most of the work around GSGP to date has focused on real-world applications [15] and on improving the search operators for the originally proposed domains (Arithmetic, Boolean, Classifiers). Relatively little attention has been paid to using the theoretical framework behind GSGP for the principled design of search operators for other domains.

Recursion is a key construct in computer programs. There have been several attempts to evolve recursive programs [1, 3, 5, 11, 16, 18]. However, GP has been found badly suited to this [17]. A major challenge is that GP operators are highly disruptive when applied to recursive programs, because small changes in the code of a recursive program cascade through the recursion, amplifying the difference in behaviour. As a result, fitness values between parent and offspring programs may vary immensely, giving rise to highly discontinuous fitness landscapes.

In this paper, we embrace the challenge of designing search operators that provably see a unimodal landscape when evolving recursive programs. Our aim is to provide theoretical insights about how to design good semantic-aware search operators for recursive domains, by studying a small and well-defined domain – a kind of 'onemax' of recursive programs. We extend the GSGP framework to the domain of recursive programs that map variable-length Boolean lists to Boolean values. For this domain, we use the theoretical observations to guide the design of search operators that are *provably efficient* in the training phase and produce *provably correct* programs, attaining so perfect generalization.

2 GEOMETRIC OPERATORS FOR RECURSION

In this section, we design geometric semantic search operators for recursive Boolean problems, so that GP with these operators will be guaranteed to be efficient in the *training* phase.

2.1 Naive approach

The conventional geometric semantic search operators for the Boolean domain introduced in [8] are:

- Crossover: $T_3 = (T_1 \wedge TR) \vee (\neg TR \wedge T_2)$, where TR is a random program.
- Mutation: $TM = T \vee M(I)$ with probability 0.5 and $TM = T \wedge \neg M$ with probability 0.5 where $M(I)$ is a random minterm of all variables in program input I.

When applied to Boolean programs of fixed input arity, i.e. $\mathbb{B}^n \to \mathbb{B}$, these operators are geometric under the Hamming distance on the output vectors, i.e., mutation changes a single entry of the semantic vector, crossover produces offspring with intermediate semantics of parents, and the fitness landscape seen by a GP algorithm using these operators is unimodal with constant slope.

GECCO '17, Berlin, Germany

978-1-4503-4920-8/17/07…$15.00
DOI: 10.1145/3071178.3071266

By extending the discourse to recursive programs, we move to the domain of programs with signature $\mathbb{BL} \to \mathbb{B}$ (*BL domain* for short), where $\mathbb{BL}$ is the space of all lists of Booleans. In this domain, it becomes convenient to rewrite the original operators using the `if-then-else` construct:

- Crossover: T_3 = `if` TR `then` T_1 `else` T_2
- Mutation: TM = `if` $M(I)$ `then` RC `else` T, where RC is a random constant `true` or `false`, each with prob 0.5.

Note that these are the geometric semantic operators proposed in [8] *for the domain of classifiers*. This is not incidental, as classifiers form a super-domain of Boolean expressions, which can be thought of as binary classifiers fed with binary variables.

The random Boolean expression `TR` used originally in crossover naturally extends to a randomly generated (and possibly recursive) program `RP` in the BL domain. For mutation, the original random minterm `M` cannot be directly applied to program input, which is a variable-length list rather than a fixed-size set of named input variables. To make M behave like a minterm for the BL domain, i.e. return `true` only for a single input and `FALSE` otherwise, we implement it as `eq(I, RList)`, where `RList` is a random Boolean list of length smaller or equal than the maximum length of input vectors in the training set, and the primitive `eq` tests its arguments for equality. We then define the following operators:

- SC(P$_1$,P$_2$) = `if RP then` P$_1$ `else` P$_2$
- SM(P) = `if eq(I,RList) then RC else P`

`RP`, `RList` and `RC` are drawn uniformly and independently in each application of SC and SM.

At this point it becomes essential to clarify how search operators affect recursion. To make sure that SC and SM preserve recursive calls in parent programs, we use generic call `self`, so that e.g. the factorial function can be expressed as `fact(n) = if n=0 then 1 else n` $\times$ `self(n-1)`. The alternative approach of calling `fact` explicitly would result in an offspring that calls its parents rather than itself.

2.2 Scaffolded GSGP operators

Though the above extension seems to naturally generalize GSGP to handle variable-length inputs, it is naive in ignoring the effects of recursive calls on program semantics. As a matter of fact, SC and SM *do not induce a unimodal fitness landscape* because the recursion interferes with the intended outcome of the functional application of GSGP search operators to recursive parents.

Proof sketch: Consider program of the form `p(i) = g(i, self(i-1))`, in which `self`, as per above convention, refers to `p`. Applying SM to `p` results in `o(i) = SM(g(i, self(i-1)))`, in which `self` refers to `o`. This is not equivalent to the intended offspring `o(i) = SM(p(i))` with `p(i)=SM(g(i, p(i-1)))`, as `SM` is not inside the body of resulting recursive program. An analogous reasoning holds for semantic crossover.

This non-geometric character SM and SC can be seen as a special case of the more general *brittleness* of recursive programs, i.e. that a modification introduced by a search operator propagates through the stack of recursive calls and completely changes the behavior of a program (even though the change might have been intended to be minor). To address this brittleness in the general, non-semantic context, Moraglio et al [10] proposed *scaffolding*, a technique that substitutes the recursive calls with calls to the (unknown) target program, using the fitness cases as a surrogate for it. This allows evolving recursive programs as if they were non-recursive. Conceptually, scaffolding replaces the calls of the form `self(i-1)` with `t(i-1)`, where `t` is the *target function*, i.e. `t(i)` returns the desired output for input `i`. `t` can be seen a *partial function* that defines the desired behavior of program on the training examples only.

It should be clear at this point that we denote the argument of recursive call by `i-1` mostly for clarity and conformance with the earlier example of factorial. The way in which that argument is formed depends on the specific domain. Crucially, scaffolding allows in recursive calls only arguments that are present in the training set (i.e. belong to the domain of the partial target function `t`). In other words, for a given input `i` all its 'predecessors' that can be generated by the argument to recursive calls have to be present in the training set. This assumption implies certain consequences that we will come back to later.

With scaffolding, GSGP operators cause no interference with recursive calls, inducing so unimodal landscape and becoming efficient operators for training recursive programs.

Claim 1: Operators SC and SM are geometric under scaffolding.

Proof sketch: Consider program `p(i) = g(i, self(i-1))`. When scaffolded, `p` becomes `p'(i) = g(i, t(i-1))`. Applying SM to `p` leads to `o'(i) = SM(g(i, t(i-1)))` which is equivalent to `o'(i) = SM(p'(i))`. Therefore, SM has the desired effect of modifying the output vector of `p` at exactly one position. Analogous reasoning holds for SC. As a result, there is no interference with search operators, as search takes place in the 'scaffolded function space'. Once the optimum function `h(i,t(i-1))` in the scaffolded space is found, it can be 'descaffolded' to obtain optimal recursive function `h(i)=h(i,h(i-1))`.

Claim 2: scaffolded semantic hill-climber finds the optimum (with zero training error) in $m \log(m)$ where m is the size of the training set, for *any* target problem.

Proof sketch: The semantic mutation corresponds in the semantic space to a bit-flip mutation on a binary string of size m, i.e., the output vector on the function undergoing mutation. The hill-climber takes then $m \log(m)$ evaluations to hit the optimum (well-known runtime result [4]).

3 DOMAIN LANGUAGE AND TARGET CLASS

Domain language: There are many programming languages that can express programs with signature $\mathbb{BL} \to \mathbb{B}$. When designing the language for this study, we aimed at a minimal set of instructions that is expressive enough to (i) represent possibly many functions with that signature and (ii) represent the semantic operators concisely. Concerning (ii), note that the semantics of SM and SC need to be implemented in the programming language of consideration, in order to provide for closure.

We chose a minimal LISP instruction set augmented with elementary Boolean operators, the recursive call `self`, and the `eq` operator (Fig. 1). Defining and calling additional functions is not allowed, so the language is not Turing-complete, and not all $\mathbb{BL} \to \mathbb{B}$ functions are expressible. Nevertheless the language allows implementing many useful and interesting functions, from generalized 2-ary Boolean operators like `and` and `or` to more sophisticated concepts like testing whether the values in the input list form some repetitive pattern. The presence of two types requires strongly-typed GP systems, where programs created at random and programs

```
B ::= true | false | not(B) | and(B,B) |
     | or(B,B) | ite(B,B,B) | head(L) | empty(L)
     | eq(L,L) | self(L)
L ::= const-list | I | tail(L) | cons(B,L)
     | ite(B,L,L)
```

Figure 1: The grammar defining the set of considered programs. `const-list` is a constant list of Booleans, `I` is the input list, `self` implements the recursive call, `eq` tests two lists for equality, and `ite` stands for `if-then-else`. The starting symbol of the grammar is `B`.

produced by search operators are guaranteed to be type-correct. Note that *SM* and *SC* applied to type-correct parent programs produce type-correct offspring programs, as a consequence of being semantically well-defined.

Target functions: We consider a family of target functions with tunable difficulty that allow expressing common recursive functions in the adopted programming language. They generalise the well-known `fold` function defined as:

```
fold(I) = ite(empty(I), bc0,
  bf2(head(I), self(tail(I)))),
```

where $\texttt{bf}_2$ is a 2-ary Boolean function, and $\texttt{bc}_0$ is a Boolean constant. List-wise `and`, list-wise `or` and list-wise `parity` belong to this class, which we term `1-fold` functions. This class can be generalised to a $k-$`fold` class that includes all functions with recursion order up to k. For instance, the blueprint for `2-fold` functions is:

```
fold2(I) = ite(empty(I), bc0,
  ite(eq(I,[false]), bc1,
    ite(eq(I,[true]), bc2,
      bf4(head(I), self(tail(I)),
        head(tail(I)), self(tail(tail(I)))))))
```

where $\texttt{bf}_4$ is a 4-ary Boolean function, $\texttt{bc}_i$ are Boolean constants, and `[false]` denotes a list containing one element `false`. Higher order of recursions are obtained by recursive calls with successive tails of the input list. The so defined order of recursion is a natural generalisation of the concept of the order of recursion in, e.g., recursive formulas. For instance, factorial has the order of recursion one because `fact(n)=g(fact(n-1))`, while the Fibonacci function has order two because `fib(n)=g(fib(n-1),fib(n-2))`.

Handling run-time errors: In the BL domain, syntactical correctness of a program does not guarantee its error-free execution. To deal with errors, we devise a penalization approach that has a natural semantic interpretation and, importantly, *does not alter the unimodality of the fitness landscape*. When a program applies `head` or `tail` to an empty list, or applies `self` to a list that is not shorter than its argument `I`[1], we assume that its output for that input is a designated special value `err`. As a consequence, program semantics is a ternary vector of three symbols: `true`, `false`, and `err`. The definition of semantic distance remains unchanged, i.e. it is the Hamming distance, and so does fitness, i.e. Hamming distance of program's semantics from the target semantics. Because the target semantics does not ever contain `err`, each error for a training example (test) results in a unit penalty. This error handling allows us to apply semantic operators also to erroneous programs and reason about the effects of such applications. Crucially, the operators remain geometric and so handle errors seamlessly.

[1]Recursive call with an argument list that is not shorter than `I` does not need to lead to infinite recursion, but preventing such cases is essential for scaffolding.

4 EXPERIMENT 1

We empirically verify the properties of GSGP in the naive and scaffolded variant on the *BL* domain. Benchmarks are parameterized by the order k of the target k-fold function and the length n of lists in the training set. The training set contains all $2^{n+1}-1$ lists of length up to n. A particular instance of target function is constructed by randomly drawing the 2^k-1 random constants $\texttt{bc}_i$ that determine the response to input lists of lengths $< k$ and the 2^k entries in the truth table of the $\texttt{bf}_{2k}$ function that aggregates the leading elements with the recursive calls on tails. This is done independently for each run, so that each configuration faces the same instances of k-fold functions.

The compared configurations of synthesis methods span three dimensions: search algorithms, search operators, and scaffolding. They all start with program trees initialized by the `RandProgram(type)` function, which recursively traverses the derivation tree of the grammar in Fig. 1 from the starting symbol of type `type` (B for initialization) and randomly picks the expressions from the right-hand sides of productions. Once this process reaches 4 in resulting program tree, the algorithm starts picking productions that immediately lead to terminals whenever possible. If the depth exceeds 5, *RandProgram* terminates, discards the tree, and starts anew. The constant lists (`const-list`) are drawn uniformly from the training set. There is no limit on the size nor the depth of program trees.

Search algorithms: We compare the population-based generational evolutionary algorithm (EA) and a single-point stochastic hill climber (SHC). In EA, we evolve a population of 1000 programs for 100 generations, selecting the parents using tournament selection with pool size 7, and breeding new programs with mutation or crossover in proportions 50 : 50. In SHC, there is only one working solution: in each iteration, a mutation operator is applied to it, and if the offspring is better, it replaces the working solution. This cycle is repeated up to 100,000 times.

Search operators: We compare GSGP operators *SM* and *SC* defined in Section 2 (GSGP), standard GP operators (GP), and random search (RS). Random programs `RP` in *SC* are obtained by calling `RandProgram(B)`, and they may include recursive calls `self`. GP employs typed variants of subtree-replacing mutation and subtree-swapping crossover. Mutation picks a random node n in the parent tree and replaces the subtree rooted in n with a subtree generated by calling `RandProgram(type(n))`. Crossover draws a random node n_1 in the first parent program, and builds the list l of same-type nodes in the second parent. If l is empty, it draws n_1 again and retries. Otherwise, it draws n_2 from l and exchanges the subtrees rooted in n_1 and n_2. The retries are guaranteed to terminate, as both parent trees always feature at least one node of type (B), i.e. the root node, and root nodes are permitted to be swapped. RS uses one search operator that discards the parent program and draws a new program by calling `RandProgram(B)`. RS is used only in combination with SHC and without scaffolding.

Scaffolding: Calling `self(L)` with an argument list `L` longer than `I` interrupts execution and returns the special `err` value introduced in Section 3. For `L`s shorter than `I`, `self(L)` returns the corresponding desired output for `L` from the training set in configurations with

enabled scaffolding, or simply performs recursive call in remaining configurations.

Table 1 presents the performance indicators of particular configurations obtained from 50 runs of each configuration on each benchmark for $k \in [1,4]$ and $n \in [1,5]$. A run is considered successful if it yields a program that produces correct outputs for all training examples. The number of evaluations, generalization error, and graph size concern best-of-run programs and are averaged *over successful runs only*; the blank table cells mark configurations where no run succeeded. *Evaluations* presents the $\log_{10}$ of the number of evaluations elapsed. *Generalization error* is the percentage of lists of lengths in $[n+1, n+2]$ for which the program produced incorrect output (or `err`). *Graph size* is the number of unique nodes in a program, i.e. the size of program tree when 'compressed' to a graph. Graph size is more appropriate than size of program tree, as GSGP operators produce programs that refer to (call) the same ancestor programs multiple times. It is also consistent with natural implementation in functional programming languages (used in our software framework), where programs are immutable and thus there is no need for cloning code pieces.[2] Functional implementation forms a natural alternative to the efficient implementation introduced by Vanneschi et al [14].

5 ANALYSIS 1

In this section, we confront the experimental results with expectations grounded in the theory.

For the *success rate*, from the theory we expect: (i) semantic HC and *with* scaffolding and semantic EA with scaffolding to have 100% success rate as the landscape seen is unimodal and the cap on the number of evaluations is generously large; (ii) semantic HC *without* scaffolding to have success rate lower than 100% as the landscape seen is not guaranteed to be unimodal and may contain local optima; (iii) in general all configurations with scaffolding to attain higher success rates, as they see a smoother landscape; (iv) EA configurations to attain higher success rate than HC, as having multiple working solutions lessens the risk of getting stuck in local optima. The experiments completely confirm (i) and (iv). (ii) is also confirmed, though the landscape seen using the naive semantic operators without scaffolding is often unimodal. Concerning (iii), the presence of scaffolding does not seem to improve the success rate except for the case of the semantic HC. In the previous work [10], scaffolding was shown to be helpful on average on a large suite of problems, so this may be due to the specific class of problems studied here.

For the *number of fitness evaluations to reach the optimum*, from the theory we expect: (i) the runtime of semantic HC with scaffolding to grow slowly with n (more precisely $m \log m$ where m is the number of fitness cases), and to not depend on k; (ii) semantic HC with scaffolding to converge faster than EA with scaffolding, as on a unimodal landscape HC converges faster than population-based algorithms; (iii) configurations with semantic operators (SGP) to scale much better than those with traditional operators (GP), as the former see a unimodal landscape. Experiments confirm (i), (ii) and (iii). In particular, semantic HC with and without scaffolding is very quick and scales much better than the other algorithms.

For *generalization error*, from the theory we expect: (i) semantic HC to not generalise well, as *SM* is designed with the sole purpose of making training efficient and will tend to memorise the training set without making use of recursive calls; (ii) semantic EA to generalise better than semantic HC, as *SC* tends to introduce new recursive calls in the offspring; also, semantic EA should generalise better on larger training sets, as this favours offspring using recursive calls to fit the training set rather than memorising one entry at a time; (iii) operators using semantics and scaffolding to lead to worst generalisation than standard GP, because they have not been designed with generalization in mind. Experiments confirm all these expectations. In particular, HC does not generalize better for increasing n, which confirms that it does memorise the training set. Note also how for sufficiently large n relative to k, semantic EA always achieves perfect generalisation, which implies making proper use of recursive calls. In general, population-based algorithms perform similarly whether scaffolded or not, and whether semantic or not. Their generalization error improves with larger n and deteriorates for larger k. For HC, standard GP generalises better than semantic HC.

For the *graph size* of zero-error programs, from the theory we expect: (i) the size of the final solution to be linear in function of the number of fitness evaluations for semantic approaches; (ii) larger sizes for population-based semantic approaches as they use crossover; (iii) smaller program sizes for standard GP than for SGP as SGP solutions grow inherently steadily in size while traditional GP do not have such a systematic bias. Experiments (not shown) confirm these expectations. SGP and GP produce programs of generally comparable sizes, but traditional GP tends to suffer less from program growth.

6 DESIGN OF SEARCH OPERATORS FOR GOOD GENERALISATION

In this section, we design new geometric semantic search operators that not only scale provably well in convergence on the training set, but also guarantee such programs to *generalise provably well* on *all* (infinitely many) unseen inputs.

6.1 Requirements for Good Generalisation

PAC-learning [13] is a theoretical framework for deriving guarantees on generalisation for classes of Boolean functions. It cannot be directly applied to synthesis of correct recursive Boolean programs, as the generalisation sought there is perfect, rather than only probably approximately correct. However, PAC-learning brings an important lesson about generalisation in general: (i) provably good generalisation can be achieved on suitably small function classes, and with a suitably large training set; (ii) generalisation is a property of *only* the class of problem considered and the size (and distribution) of the training examples, and not of the training algorithm used. The only requirement on the training algorithm is to (efficiently) find a function *within the problem class* with zero-error on the training set.

By analogy to fixed-length program semantics for fixed-arity programs, let *infinite semantics* of program P be the (infinite) vector (sequence) of outputs produced by P for all input lists, ordered w.r.t. increasing length (and arbitrarily otherwise).

Claim: 1) For the class of $k-$`fold` functions for any given k, a prefix of *finite* length m_k* of infinite semantics is sufficient to uniquely identify any function in the class up to functions with the same infinite semantics. 2) Any function in this class that has zero error

[2] https://github.com/amoraglio/GSGP, https://github.com/kkrawiec/swim

Table 1: Performance indicators for the GSGP and GP operators, compared to random search RS

				Fold1					Fold2					Fold3					Fold4				
			$n=$	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
Success rate	GP	EA		100	100	98	96	82	100	96	54	40	18	100	98	62	28	12	100	100	76	36	8
			Scaff.	100	100	98	84	90	100	96	58	36	10	100	96	56	32	10	100	92	44	30	6
		HC		100	90	84	80	68	100	98	38	18	4	100	94	28	0	0	100	94	24	2	0
			Scaff.	100	84	76	70	68	100	94	36	12	6	100	92	18	2	0	100	90	22	2	0
	SGP	EA		100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
			Scaff.	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
		HC		100	100	98	100	98	100	100	98	98	96	100	100	98	100	100	100	98	98	98	96
			Scaff.	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
	RS			100	80	72	70	70	100	66	4	4	4	100	40	0	0	0	100	52	0	0	0
Evaluations	GP	EA		3.0	3.6	3.9	4.1	4.1	3.0	3.8	4.3	4.6	4.7	3.1	3.9	4.4	4.6	4.8	3.0	4.0	4.5	4.7	4.9
			Scaff.	3.0	3.7	3.9	3.9	4.0	3.0	3.9	4.4	4.5	4.4	3.0	4.0	4.4	4.6	4.8	3.1	4.0	4.3	4.6	4.9
		HC		3.3	4.9	4.9	4.8	4.6	3.3	5.0	5.5	5.5	0.7	3.1	5.2	5.6			3.0	5.2	5.6	5.9	
			Scaff.	3.3	4.9	4.9	4.3	4.6	3.3	5.1	5.6	5.6	5.0	3.1	5.2	5.6	5.9		3.0	5.2	5.6	5.9	
	SGP	EA		3.0	3.2	3.5	3.7	4.0	3.0	3.4	3.8	4.1	4.4	3.0	3.4	3.8	4.2	4.5	3.0	3.4	3.8	4.2	4.5
			Scaff.	3.0	3.2	3.4	3.5	3.7	3.0	3.4	3.8	4.1	4.3	3.0	3.4	3.8	4.2	4.5	3.0	3.4	3.8	4.2	4.5
		HC		1.1	1.4	1.9	2.3	2.7	1.0	1.4	1.9	2.3	2.7	0.9	1.5	1.9	2.4	2.7	1.0	1.4	1.9	2.3	2.7
			Scaff.	1.1	1.4	1.9	2.3	2.7	1.0	1.4	1.9	2.3	2.7	0.9	1.5	1.9	2.4	2.7	1.0	1.4	1.9	2.3	2.7
	RS			2.5	4.7	4.6	4.4	4.7	2.6	5.3	0.6	0.6	0.6	2.8	4.9				2.8	5.2			
Generalization	GP	EA		26	12	9	4	3	45	39	32	27	18	53	49	47	44	37	52	50	50	48	49
			Scaff.	26	10	5	1	1	45	39	32	26	12	54	48	48	45	37	52	49	48	48	48
		HC		25	9	6	0	1	46	38	27	9	0	53	48	51			52	48	53	58	
			Scaff.	25	8	2	0	0	46	37	27	12	0	53	47	53	27		52	49	51	58	
	SGP	EA		28	12	10	9	8	46	39	36	29	25	53	49	47	45	42	52	50	48	50	49
			Scaff.	28	12	4	2	2	46	38	33	24	17	53	49	47	43	40	52	51	49	48	49
		HC		69	62	64	64	64	55	56	55	57	56	65	62	62	58	58	56	58	58	55	55
			Scaff.	69	63	64	64	64	55	56	55	57	56	65	62	62	58	58	56	58	58	56	56
	RS			27	6	0	0	0	44	36	0	0	0	53	48				51	50			

on examples corresponding to prefix elements generalises perfectly on *any* unseen input. This result holds for *any* training algorithm.
Proof sketch: 1) There is a finite number of functions belonging to a given k-fold class. Consider the set S of infinite semantics of all those programs. Each pair of them is either equal or different. For each different pair of semantics $s_1, s_2 \in S$ consider the shortest prefix that discriminates them. Its length must be finite, otherwise s_1 and s_2 cannot be different. Now let m_k* be the maximum length of such prefixes over each pair of semantics in S. By construction the prefix of length m_k* discriminates any two semantics in S. If two semantics in S have the same prefix of length m_k*, they must be identical i.e., $\forall s_1, s_2 \in S : s_1[1 : m_k*] = s_2[1 : m_k*] \Rightarrow s_1 = s_2$.
2) Note that the unknown target function t also belongs to the k-fold class, so its infinite semantics $s_t \in S$. As per 1), if any k-fold function f with infinite semantics s_f coincides with s_t on the prefix of length m_k*, they cannot differ on the remaining elements, and it must hold that $s_f \equiv s_t$, and so f must have zero generalisation error on all examples.

For prefix lengths $l < m_k*$, we expect generalization error to decrease with l, as longer prefixes are more selective. This may be useful if m_k* is reasonably small. For the 1-fold class, we enumerated all output sequences and determined that $m_1* = 15$.

6.2 k-fold GSGP Operators

The practical upshot of the arguments brought in the previous section is that by (i) evaluating programs on a sufficiently large number of training cases *and* (ii) performing search in the space of k-fold functions only, the resulting program is guaranteed to generalize perfectly if it achieves zero error on the training set. The former is beyond our control in real-world settings, so we leave it open. The latter depends on the design of initialization and search operators. Therefore, in the following we refine the scaffolded GSGP search operators to make sure that the search takes place in the k-fold space only.

To begin with, we assume that the initial population contains only k-fold programs. This can be easily achieved using appropriate initialization operator, e.g. the procedure for generating k-fold target functions described in Section 4. Then, we propose the following two mutation operators:

$SM1_k$(P) = ite(eq(I,rl), bc, P)
$SM2_k$(P) = ite(len(I)≥k, SMB(P), P)

where bc is a Boolean constant, and rl is sampled from the set of all Boolean lists of length up to $k-1$, and SMB is the semantic (minterm) mutation for Boolean expressions. We combine $SM1_k$ and $SM2_k$ into one k-fold mutation operator SM_k by using the former with probability $n_c/(n_c+n_f)$ and the latter with probability $n_f/(n_c+n_f)$, where $n_c = 2^k - 1$ and $n_f = 2^{2k}$. We define k-fold crossover as:

SC_k(P1, P2) = ite(rf, P1, P2)

where rf is a random k-fold function.

In the following we show that SM_k and SC_k are k-fold preserving, i.e., given parents $p, p' \in \mathcal{F}_k$, the offspring $SM_k(p), SC_k(p,p') \in \mathcal{F}_k$. We show this only for $k = 1$, but the reasoning holds for any k. As shown in Section 3, functions in $\mathcal{F}_1$ can be written in the 1-fold form as fold(I) = ite(empty(I), bc, bf(head(I), self(tail(I)))), where bc is a Boolean constant and bf is a 2-ary Boolean function. If a function g can be rewritten in this form, then

$g \in \mathcal{F}_1$. It is therefore sufficient to show that given two parents in 1-fold form, the generated offspring can be rewritten in the 1-fold form.

For mutation, $SM1_k$ and $SM2_k$ take the following form for $k = 1$:

```
SM1₁(P) = ite(eq(I,[]), bc, P)
SM2₁(P) = ite(len(I)≥1, SMB(P), P)
```

where $SM1_1$ is engaged with probability 0.2 and $SM2_1$ with probability 0.8. The outcome of $SM1_1$ applied to a program f can be rewritten as:

```
SM1₁(f(I)) = ite(I=[], bc, f(I))
   = ite(I=[], bc,
      ite(I=[], bc, bf(head(I), self(tail(I))))
= ite(I=[], bc, bf(head(I), self(tail(I)))
```

The offspring can be thus written in the 1-fold form, so $SM1_1$(f) $\in \mathcal{F}_1$. For $SM2_1$:

```
SM2₁(f(I)) = ite(len(I)≥1, SMB(F(I)), F(I))
= ite(len(I)≥1,
  SMB(ite(I=[], bc, bf(head(I), self(tail(I))))),
     ite(I=[], bc, bf(head(I), self(tail(I)))))
= ite(len(I)≥1, SMB(bf(head(I), self(tail(I))), bc)
= ite(I=[], bc, SMB(bf(head(I), self(tail(I))))
```

The last expression is also in the 1-fold form, as the expression SMB(bf(head(I), self(tail(I)))) is an expression whose inputs are head(I) and self(tail(I)), as required by the 1-fold form. In this case, semantic mutation changes bf(head(I), self(tail(I)) of the parent to SMB(bf(head(I), self(tail(I))), i.e. modifies a single entry in the truth table of bf. $SM1_1$ and $SM2_1$ are thus 1-fold preserving, and so is SM_1.

For crossover SC_1 applied to a parent program f(I):

```
O(I) =SC₁(f(i)) = ite(rf(I), P1, P2)
= ite(ite(I=[], RBC, bf(head(I), self(tail(I))),
    ite(I=[], bc₁, bf_p1(head(I), self(tail(I)))),
    ite(I=[], bc₂, bf_p2(head(I), self(tail(I)))))
```

If I=[] the above expression becomes OBC = ite(RBC, bc_1, bc_2); otherwise it becomes:

```
OBF2(head(I), self(tail(I))) =
= ite(RBF2(head(I), self(tail(I)),
    bf_p1(head(I), self(tail(I)),
    bf_p2(head(I), self(tail(I)))
```

Hence the offspring is O(I) = ite(I=[], OBC, OBF2(head(I), self(tail(I)))), which is in the 1-fold form because: (i) its bc is a Boolean constant inherited from the bc of either parents depending on the value of RBC, and its bf part is of the required form as ite can be thought as a Boolean function taking three Boolean inputs, and its composition on RBF2, bf_{p1}, and (ii) bf_{p2} is a Boolean function of head(I) and self(tail(I)). Hence SC_1(f)$\in \mathcal{F}_1$.

Note that these operators are 1-fold preserving also when using scaffolding, as replacing the recursive call self with the desired output known from the training set does not affect the reasoning.

Geometric characteristics of k-fold operators: In order to analyse the k-fold operators, we introduce the notion of *intensional semantics*. The intensional semantics s_i(f) of a 1-fold function f(I)=ite(I=[], bc, bf(head(I), self(tail(I)))) is the concatenation of the random constant bc and the output vector of the bf function (i.e., dependent column of its truth table). For instance, the intensional semantics of list-wise AND, which can be written in 1-fold form as f(I)=ite(I=[], True, And(head(I), self(tail(I)))), is 1 (bc = True) followed by the output vector of And ($00 \to 0, 01 \to 0, 10 \to 0, 11 \to 1$), i.e. (10001). For $k > 1$, all random constans bc_i are obviously concatenated.

The intensional semantics s_i of a k-fold function is an alternative way of representing the function, which identifies it uniquely in $\mathcal{F}_k$. To avoid confusion, from now on we refer to the standard notion of semantics (the vector of function outputs) as *extensional semantics* and denote it with s_e.

Endowing intentional semantics with the Hamming distance creates *intentional semantic space* of k-fold functions. It should be clear from the previous section that SM_k and SC_k correspond respectively to bit-flip mutation and uniform crossover *in that space*, i.e., they are geometric in the intensional space. This however does not imply that they see a unimodal landscape, as the fitness function is based on the distance of the extensional semantic space. In the following we provide better understanding of the relationship between these two spaces.

Consider 1-fold function g(I) = ite(I=[],B0, f(head(I), g(tail(I)))). s_i(g) is ([], B0) concatenated with the truth table of f, i.e. $\{[] \mapsto B0, 00 \mapsto B1, 01 \mapsto B2, 10 \mapsto B3, 11 \mapsto B4\}$.

The extensional semantics s_e(g) is an infinitely long vector, the first elements of which are as follows:

I	g	s_e(g)
[]	B0	G1
[0]	f(0,g([])) = f(0,G1)	G2
[1]	f(1,g([])) = f(1,G1)	G3
[00]	f(0,g([0])) = f(0,G2)	G4
[01]	f(0,g([1])) = f(0,G3)	G5
[10]	f(1,g([0])) = f(1,G2)	G6
[11]	f(1,g([1])) = f(1,G3)	G7
[000]	f(0,g([00])) = f(1,G4)	G8
...		

Clearly, the entries in $s_e(g)$ depend in general not only on the entries in the intensional semantic table, but also on the previous entries in $s_e(g)$. Flipping a single bit in $s_i(g)$ may affect arbitrarily many (even infinitely many) bits in $s_e(g)$, as the changes propagate via recursive calls. Thus, 'fixing' an entry in $s_i(g)$ (which is the general intent of GSGP search operators) does not necessarily lead to 'fixing' entries in $s_e(g)$, unless previous entries in the table are also correct. Fixing a single bit in $s_i(g)$ towards the intensional representation of the optimum may result in 'unfixing' more entries in $s_e(g)$ towards the extensional semantics of the target, worsening the fitness. The extensional semantic space may thus feature local optima.

Scaffolding changes this picture, replacing the recursive call with the call of the actual target function t, so that g becomes g'(I) = ite(I=[], B0, f(head[I], t(tail[I]))), which in turn leads to the following extensional semantics s_e(g'):

I	g'	t	s_e(g')
[]	B0	T1	G1'
[0]	f(0,t([])) = f(0,T1)	T2	G2'
[1]	f(1,t([])) = f(1,T1)	T3	G3'
[00]	f(0,t([0])) = f(0,T2)	T4	G4'
[01]	f(0,t([1])) = f(0,T3)	T5	G5'
[10]	f(1,t([0])) = f(1,T2)	T6	G6'
[11]	f(1,t([1])) = f(1,T3)	T7	G7'
[000]	f(0,t([00])) = f(1,T4)	T8	G8'
...			

The key observation is that, as the target is fixed, the entries in $s_e(g')$ depend directly, in a non-recursive way, on the entries of

s_i(g') (which is the same as s_i(g). For instance, when t is the function that always returns 0, we have: g'([])=G1'=B0, g'([0])=G2' =f(0,0)=B1, g'([1])=G3'= f(1,0)=B3, g'([00])=G4'=f(0,0) =B1, ... Moreover, the entries in s_e(g') can be grouped into disjoint blocks, each corresponding to one entry in s_i(g') and having the same output as that entry. For instance, g'([0]) and g'([00]) both belong to the block corresponding to the intensional semantic entry f(0,0), and have the same B1 output.

This implies the existence of a relation between the distance in the intensional semantic space (SD_i) and the distance in the extensional semantic space (SD_e). For any two k-fold functions f_1 and f_2, $SD_i(f_1, f_2) = HD(s_i(f_1), s_i(f_2))$, where HD is Hamming distance. In turn, $SD_e(f_1, f_2) = HD(s_e(f_1), s_e(f_2))$, but crucially $SD_e(f_1, f_2)$ can be also expressed as a weighted HD of their intensional semantics, in which the weight of each contributing entry is the size of the corresponding block in the extensional semantics. For the fitness landscape seen from the intensional space it holds that for every point P at a distance $SD_i(P, t)$ to the target t, its fitness is $Fit(P) = SE_e(P, t) = HD_w(s_i(P), s_i(t))$. This implies that the fitness landscape as seen by the search operators in the intensional semantic space is still unimodal, as moving towards the target in the intensional space (smaller $SD_i(P, t)$) corresponds to getting a better fitness (smaller $HD_w(s_i(P), s_i(t))$). However, it does not have a constant gradient (which is the case for the traditional GSGP operators). Nevertheless, when using an ordinal selection scheme (i.e., such that depends only on the order of fitness values, like tournament selection), this landscape is effectively equivalent to a unimodal landscape with a constant gradient.

In summary, we have shown above that the k-fold search operators SM_k and SC_k not only provide closure for the $\mathcal{F}_k$, but also induce a unimodal fitness landscape when combined with scaffolding. Limited space allowed us to present this only for $k = 1$, but the above reasoning elegantly generalizes to $k > 1$. As a consequence, these operators and scaffolding ensure provably efficient training and perfect generalisation. A search process starting from a population of functions from $\mathcal{F}_k$ and using SM_k and SC_k must produce a solution which is also in $\mathcal{F}_k$. Provided the target is also in $\mathcal{F}_k$, and the training set is sufficiently large (m_k* or more training examples), the solution with zero-error of the training set is guaranteed to have perfect generalisation. The landscape is unimodal, so the optimum can be found efficiently. For a stochastic hill-climber, the runtime is $l_i \log l_i$, where $l_i = 2^k - 1 + 2^{2k}$ is the length of the intensional semantics vector (in contrast the runtime of random search is exponential in l_i).

7 EXPERIMENT 2

In this experiment we evaluate the k-fold search operators proposed in Section 6.2 on the same suite of k-fold problems as in Section 4, using analogous configurations of EA and SHC with and without scaffolding. This time however we make sure that the entire search process takes place in the space of k-fold functions $\mathcal{F}_k$. To this aim, we assume that the fold order k of benchmark's target function is known to the search algorithm. Given k, we first initialize the population with programs that implement random k-fold functions. Each such program comprises a single node (terminal) $\texttt{rf}_k$ that returns the value of that function for the current input I. The intensional semantics (truth tables) of particular instances of $\texttt{rf}_k$ are drawn in the same way as for the target functions in Experiment 1. The programs in the population are then modified using search operators SM_k and SC_k described in Section 6.2. To implement them, we extend the set of instructions from Fig. 1 with $\texttt{rf}_k$ and the terminal instruction $\texttt{len}_k$ that implements the condition $\texttt{len(I)} \geq k$, required in $SM2_k$. We also assume this time that the constant lists (the const-list terminal) are all Boolean lists shorter than k. This extended instruction set is sufficient to implement SM_k and SC_k, including the minterm operation SMB. These search operators form the configuration $SGPf_k$.

Note that, despite featuring additional instructions, the set of programs considered here is not a superset of the language defined by the grammar in Fig. 1, because instructions can be combined only as prescribed by SM_k and SC_k, and so keep search within the bounds of $\mathcal{F}_k$. Also, recursive calls can be introduced only by mutation (contrary to SGP, where only crossover was capable of that), more precisely by the SMB term. As a consequence, new recursive calls may appear also in SHC runs that use only mutation.

The baseline method is RSk, the random search in the space of k-fold functions $\mathcal{F}_k$. RSk is a single-point hill climber that in each iteration generates a random k-fold program $p = \texttt{rf}_k$, and replaces the current program with p if the fitness of p is better.

With $\texttt{rf}_k$ terminals present in instruction set, it becomes possible to find a single-node solution to a given k-fold target function benchmark. However, the odds for this decrease rapidly with k, as there are $2^{2^k - 1 + 2k}$ k-fold functions.

The remaining settings are identical to those in Experiment 1. The results of this experiment are shown in Table 2.

8 ANALYSIS 2

For *success rate*, hitting the optimum gets harder with growing k and growing n, which is particularly evident for RS. This is expected because the number of functions in a k-fold class grows exponentially with k, and the number of functions in $\mathcal{F}_k$ that match the training examples decreases with n (up to a critical number of fitness cases).

For the *number of evaluations*, the expected number of evaluations to hit the optimum in RS is the size of $\mathcal{F}_k$ divided by the number of functions in $\mathcal{F}_k$ that match all training examples. This ratio is also a measure of difficulty of the problem. Experimental results confirm that for small n and k the assumed population size of 1000 is disproportionate w.r.t. the expected performance of RS, and the optimum is found in the initial population. The results confirm also that HC (scaffolded or not) scales much better than RS for growing k and n, and that for n and k large enough EA also scales better than RS.

For *generalisation performance*, for 1-fold class, for $n \geq 3$ all search algorithms generalize perfectly, as predicted by the theory. Perfect generalisation is independent from the search algorithm, and depends only on the problem class and sufficient number of fitness cases. When the training set is not large enough to guarantee perfect generalisation, all algorithms improve their generalisation performance for growing training set size as expected, and we notice that random search generalises the best.

For *graph size* (results not shown),the solutions found by random search are always the smallest, which must be the case all candidate solutions in RSk are single-node k-fold functions $\texttt{rf}_k$. The sizes of the programs found by the other algorithms are smaller than those not using the k-fold operators.

Table 2: Performance indicators for the k-`fold` operators, compared to random search RSk

				Fold1					Fold2					Fold3					Fold4				
			$n=$	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
Succ. rate	SGPk	EA		100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
			Scaff.	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
		HC		76	86	76	80	74	100	52	32	36	44	100	100	10	0	0	100	100	100	6	0
			Scaff.	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
	RSk			100	100	100	100	100	100	100	100	96	96	100	100	100	2	0	100	100	100	0	0
Evals.	SGPk	EA		3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.3	3.6	3.8	3.0	3.0	3.4	3.7	4.0	3.0	3.0	3.4	3.7	4.0
			Scaff.	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.3	3.6	3.8	3.0	3.0	3.4	3.7	4.0	3.0	3.0	3.4	3.7	4.0
		HC		1.1	1.3	1.3	1.3	1.3	1.6	1.8	2.1	2.1	2.2	2.2	2.5	2.5			2.9	3.0	3.1	3.3	
			Scaff.	1.1	1.2	1.2	1.2	1.2	1.6	1.9	2.0	2.0	2.0	2.2	2.5	2.6	2.7	2.7	2.9	3.0	3.1	3.2	3.3
	RSk			0.9	1.4	1.4	1.4	1.4	0.8	2.1	3.9	5.0	5.2	1.0	2.1	4.6	5.5		1.0	2.0	4.6		
Gener.	SGPk	EA		19.0	0.0	0.0	0.0	0.0	49.0	35.8	30.1	22.4	14.1	48.3	48.5	49.2	47.5	45.9	49.8	50.5	48.4	51.2	49.8
			Scaff.	19.0	0.0	0.0	0.0	0.0	49.0	35.8	29.9	21.4	13.2	48.3	48.5	49.2	47.1	46.3	49.8	50.5	48.4	50.5	50.1
		HC		29.2	3.0	0.0	0.0	0.0	52.2	41.2	31.5	7.1	3.1	51.2	50.4	45.0			49.5	50.9	49.8	49.0	
			Scaff.	30.0	4.1	0.0	0.0	0.0	52.2	42.3	26.5	8.1	2.7	51.2	50.4	46.5	41.0	29.8	49.5	50.9	49.8	48.6	46.9
	RSk			22.0	0.0	0.0	0.0	0.0	48.2	38.6	16.4	3.5	0.6	49.8	48.9	43.5	44.8		49.7	52.2	48.8		

9 CONCLUSIONS AND FUTURE WORK

Evolving recursive programs effectively is one of the long-standing challenges for GP and the key to its ultimate goal of evolving complex, fully-fledged programs from examples. In this paper, we made first steps towards designing *provably efficient* search operators for recursive domains that see a *unimodal fitness landscape* and are guaranteed to find the *correct program*. These three properties have significant implications for theory and practice of GP, and make this new line of investigation very exciting. The theoretical concepts and methodology introduced here for the specific case of recursive Boolean programs have in principle much wider applicability. There are several interesting lines of future investigation: extending this framework to arithmetic recursive domains and investigate the potential issues resulting from moving to continuous domains; extension to functions that map multiple Boolean lists to Boolean lists, a much richer class of functions (requires more complex notion of program semantics for vectors of lists); investigating classes of functions that capture traditional LISP programs manipulating nested lists of symbols (i.e., tree-like structures), and so involve richer data structures that are inherently recursive; last but not least, extending the framework to Turing-complete programs.

Acknowledgment K. Krawiec acknowledges support from grant 2014/15/B/ST6/05205 funded by the National Science Centre, Poland.

REFERENCES

[1] Alexandros Agapitos and Simon M. Lucas. 2006. Evolving Efficient Recursive Sorting Algorithms. In *Proceedings of the 2006 IEEE Congress on Evolutionary Computation*, Gary G. Yen, Lipo Wang, Piero Bonissone, and Simon M. Lucas (Eds.). IEEE Press, Vancouver, 9227–9234.

[2] Lawrence Beadle and Colin Johnson. 2008. Semantically Driven Crossover in Genetic Programming. In *Proceedings of the IEEE World Congress on Computational Intelligence*, Jun Wang (Ed.). IEEE Computational Intelligence Society, IEEE Press, Hong Kong, 111–116.

[3] Scott Brave. 1996. Evolving Recursive Programs for Tree Search. In *Advances in Genetic Programming 2*, Peter J. Angeline and K. E. Kinnear, Jr. (Eds.). MIT Press, Cambridge, MA, USA, Chapter 10, 203–220. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.3.3005

[4] Stefan Droste, Thomas Jansen, and Ingo Wegener. 2002. On the analysis of the (1+ 1) evolutionary algorithm. *Theoretical Computer Science* 276, 1 (2002), 51–81.

[5] John R. Koza. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA. http://mitpress.mit.edu/books/genetic-programming

[6] Krzysztof Krawiec and Pawel Lichocki. 2009. Approximating geometric crossover in semantic space. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, Guenther Raidl et al (Eds.). ACM, Montreal, 987–994.

[7] Nicholas Freitag McPhee, Brian Ohs, and Tyler Hutchison. 2008. Semantic Building Blocks in Genetic Programming. In *Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008 (Lecture Notes in Computer Science)*, Michael O'Neill et al (Eds.), Vol. 4971. Springer, Naples, 134–145.

[8] Alberto Moraglio, Krzysztof Krawiec, and Colin G. Johnson. 2012. Geometric Semantic Genetic Programming. In *Parallel Problem Solving from Nature, PPSN XII (part 1) (Lecture Notes in Computer Science)*, Carlos A. Coello Coello et al (Eds.), Vol. 7491. Springer, Taormina, Italy, 21–31.

[9] Alberto Moraglio, Andrea Mambrini, and Luca Manzoni. 2013. Runtime Analysis of Mutation-Based Geometric Semantic Genetic Programming on Boolean Functions. In *Foundations of Genetic Algorithms*, Frank Neumann and Kenneth De Jong (Eds.). ACM, Adelaide, Australia, 119–132.

[10] Alberto Moraglio, Fernando Otero, Colin Johnson, Simon Thompson, and Alex Freitas. 2012. Evolving Recursive Programs using Non-recursive Scaffolding. In *Proceedings of the 2012 IEEE Congress on Evolutionary Computation*, Xiaodong Li (Ed.). Brisbane, Australia, 2242–2249.

[11] Lee Spector, Jon Klein, and Maarten Keijzer. 2005. The Push3 execution stack and the evolution of control. In *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, Hans-Georg Beyer et al (Eds.), Vol. 2. ACM Press, Washington DC, USA, 1689–1696.

[12] Nguyen Quang Uy, Nguyen Xuan Hoai, Michael O'Neill, R. I. McKay, and Edgar Galvan-Lopez. 2011. Semantically-based crossover in genetic programming: application to real-valued symbolic regression. *Genetic Programming and Evolvable Machines* 12, 2 (June 2011), 91–119.

[13] L. G. Valiant. 1984. A Theory of the Learnable. *Commun. ACM* 27, 11 (Nov. 1984), 1134–1142.

[14] Leonardo Vanneschi, Mauro Castelli, Luca Manzoni, and Sara Silva. 2013. A New Implementation of Geometric Semantic GP and its Application to Problems in Pharmacokinetics. In *Proceedings of the 16th European Conference on Genetic Programming, EuroGP 2013 (LNCS)*, Krzysztof Krawiec et al (Eds.), Vol. 7831. Springer Verlag, Vienna, Austria, 205–216.

[15] Leonardo Vanneschi, Sara Silva, Mauro Castelli, and Luca Manzoni. 2013. Geometric Semantic Genetic Programming for Real Life Applications. In *Genetic Programming Theory and Practice XI*, Rick Riolo, Jason H. Moore, and Mark Kotanchek (Eds.). Springer, Ann Arbor, USA, Chapter 11, 191–209.

[16] P. A. Whigham and R. I. McKay. 1995. Genetic approaches to learning recursive relations. In *Progress in Evolutionary Computation*, Xin Yao (Ed.). Lecture Notes in Artificial Intelligence, Vol. 956. Springer-Verlag, 17–27.

[17] John R. Woodward and Ruibin Bai. 2009. Why evolution is not a good paradigm for program induction: a critique of genetic programming. In *GEC '09: Proceedings of the first ACM/SIGEVO Summit on Genetic and Evolutionary Computation*, Lihong Xu et al (Eds.). ACM, Shanghai, China, 593–600.

[18] Tina Yu and Chris Clack. 1998. Recursion, Lambda Abstractions and Genetic Programming. In *Genetic Programming 1998: Proceedings of the Third Annual Conference*, John R. Koza et al (Eds.). Morgan Kaufmann, University of Wisconsin, Madison, Wisconsin, USA, 422–431. http://www.cs.mun.ca/~tinayu/index_files/addr/public_html/Recursion.pdf