# Analysis of Semantic Modularity for Genetic Programming

Krzysztof KRAWIEC Bartosz WIELOCH *

**Abstract.** In this paper we analyze the properties of *functional modularity*, a concept introduced in [14] for detecting and measuring modularity in problems of automatic program synthesis, in particular by means of genetic programming. The basic components of functional modularity approach are *subgoals* – entities that embody module's semantic – and *monotonicity*, a measure for assessing subgoals' potential utility for searching for good modules. For a given subgoal and a sample of solutions decomposed into *parts* and *contexts* according to module definition, monotonicity measures the correlation of distance between semantics of solution's part and the fitness of the solution. The central tenet of this approach is that highly monotonous subgoals can be used to decompose the task and improve search convergence. In the experimental part we investigate the properties of functional modularity using eight instances of problems of Boolean function synthesis. The results show that monotonicity varies depending on problem's structure of modularity and correctly identifies good subgoals, potentially enabling automatic program decomposition.

**Keywords:** Automatic Program Synthesis, Modularity, Problem Decomposition, Evolutionary Computation, Genetic Programming

## 1 Introduction

The objective of this study is to explore a new formulation of detection and exploitation of modularity in problems of automatic program synthesis. In general, the task of automatic program synthesis may be formulated in the following way: given a programming language $\mathcal{L}$ and a set of examples $\mathcal{E}$ of desired behavior of a program, synthesize a program in $\mathcal{L}$ that reproduces that behavior. By *behavior* we mean here the output produced by the program in response to some input, thus each example in $\mathcal{E}$ is equivalent here to a pair, where the first element of the pair is the input data for

*Institute of Computing Science, Poznan University of Technology, Piotrowo 2, 60965 Poznań, Poland, {kkrawiec,bwieloch}@cs.put.poznan.pl

the program, and the second element of the pair is the desired output of the program. The ultimate goal is to reproduce the behavior given by $\mathcal{E}$ perfectly; however, in case of iterative methods, it is often necessary to know how much the actual program output diverges from the desired one, which in turns implies the need of existence of some kind of divergence measure (typically a metric).

Approaches to automatic program synthesis have originated from different branches of computational intelligence. Examples include inductive logic programming and automatic feature construction for learning from examples. The background of this particular study is genetic programming (GP), an evolution-inspired method for program synthesis. Technically, GP is a variant of evolutionary computation (EC) [11, 8, 18]. A typical GP algorithm is a randomized iterative parallel search procedure that maintains a set (population) of working solutions (individuals), each of them being a program. The programs are composed of primitives (instructions) from a predefined set and, in the most popular variant of GP introduced by Koza [12], take the form of tree, with the leaves (terminals) representing input data and constants, and the inner nodes (functions) processing the data up to the root node. In the evaluation phase of the evolutionary algorithm, the data returned by the root node is compared to the desired output and the quantitative (e.g., distance-based) outcome of this comparison, usually averaged over the examples from $\mathcal{E}$, determines the *fitness* of the solution. The fitness values of particular individuals determine the outcome of the subsequent process of *selection*, which tends to terminate the unfit individuals and allows only the more fit individuals to pass to the next stage – breeding. The breeding phase typically embraces solution recombination, in which programs swap code fragments, and mutation, in which randomly selected code fragments are randomly perturbed. This process produces offspring solutions that fill up the new population, and the artificial evolution proceeds to the next generation. The search terminates when some problem-specific stopping criteria (typically fitness level) are met.

It should be emphasized that due to the complexity of program synthesis task, most of GP research does not involve off-shelf imperative programming languages. Rather than that, the aim is typically to evolve expressions (algebraic, logic, etc.), which are usually free of conditional statements and loops. This holds also for the study reported here.

State-of-the-art research demonstrates that GP, similarly to other approaches to automated program synthesis, fails to overcome an important challenge: scalability. Canonical GP algorithms perform well as long as the number of independent variables that form the input data is low and the expression to be evolved is relatively straightforward. Things get worse as problem instances become larger and more sophisticated. This calls for an automatic decomposition of the task of program synthesis. Given a difficult program synthesis task, a decomposition method would ideally break it down into smaller subproblems, which would be easier to solve. The overall solution to the original problem could be then easily assembled from the subproblems' solutions. Delivering such a method is a far-fetched goal of our research.

In this study we propose a specific methodology that aims at this goal, which we refer to as *functional modularity*. However, due to the preliminary character of this paper, we conduct our investigations abstracting from evolutionary computation: for-

malization as well as experimental results presented here concern static, non-evolving samples of randomly generated programs. Thus, for a reader unfamiliar with GP, this study may be considered as a statistical analysis of semantics of randomly generated Boolean expressions.

The remaining part of this paper starts with a preliminary Section 2 that provides general description of modularity and reviews research on this topic. In Section 3, following [14] we motivate, define, and formalize our approach to this problem. Section 4 reports the results of experimental validation of functional modularity on eight instances of two problems of logical function synthesis. Finally, Section 5 summarizes the experimental outcomes and outlines the future research.

## 2 Defining and Exploiting Modularity

The *No Free Lunch* theorem states that no search algorithm is better than one when compared on a uniformly distributed population of all problems [23]. In this light, superiority of some algorithms to others on real-world problems indirectly demonstrates that some problems (and problem instances) are more likely to occur in practice than others. This propels the quest for properties that are common for the real-world problems (or some classes of them) and that may be exploited in the search process for the benefit of faster convergence. Examples of such properties studied in the past include fitness-distance correlation, unimodality of the fitness landscape, and modularity, the last one being of interest here.

The term 'module' is difficult to define without referring to a more specific background. There is quite firm agreement that a module is a *part* of solution (i.e., something that may be clearly delineated from the solution), such that it exhibits some form of independence (full of partial) from the remaining part of solution (often referred to as *context*). That independence is usually understood in terms of module's *contribution* to solution's performance (fitness) [22]. In an extreme case of a fully independent module, its contribution does not depend on the context, in which case the problem becomes *separable* [21] and the module may be optimized independently using *any* context. Such scenario is however unlikely in the real world, where modules and contexts are usually *interdependent*: a module contributes to the overall fitness, but its contribution depends on the context. This dependency can take on different forms and result in module's observed contribution that is non-linear, non-monotonous, or indeterministic, rendering module detection difficult. The other factor that makes it hard is the likely presence of multiple modules, which gives rise to exponential explosion (see [21] for an in-depth analysis of modularity and related topics, like compositionality, accretive evolution, and the building block hypothesis).

The ability of a search algorithm to benefit from modularity is important, because a large proportion of real-world problems turn out to have interdependent modules. Detection and proper exploitation of modularity prior to or during search may speed up convergence, prevent code bloat, and cause the evolved solutions to be more robust. But most importantly, modularity is essential for scalability, which is a particularly difficult issue for genetic programming (GP), as demonstrated in past research [15].

The strict definition of a module obviously depends on the representation of solutions that the search algorithm operates on. In genetic algorithms (GA) or evolutionary programming, where solutions are represented as vectors of variables, module has a natural interpretation of a subset of variables. In tree-based GP, it is most common to equate a module with a subtree. Methods referring to such module definition include evolutionary module acquisition [2], automatically defined functions [13], adaptive representations [19], and hierarchical genetic programming [3]. The approaches proposed in past concerned not only canonical tree-based GP, but also other representations like cartesian genetic programming [20]. However, what these previous approaches have in common is their purely *syntax-based* formulation of modularity; in this paper, on the contrary, we aim at enriching the analysis of modularity with program *semantic*.

In general, the concept of a module for optimizing a vector of variables and the concept of a module for evolving executable programs are fundamentally different. However, modularity for vector representations seems to be a good starting point for introducing our idea of functional modularity. Thus, in following we briefly summarize a recent study on modularity in GA, related to Harik's work on learning gene linkage [10] and Goldberg's competent genetic algorithms [9].

In Watson's and De Jong's formulation [21, 7], given a set of variables $V$, a module is identified with a subset of variables $M \subset V$ such that the linkage between variables in $M$ is tighter than the linkage between variables from $M$ and the variables from the context, i.e., $V \backslash M$. The definition of linkage refers to the notion of *context-optimal setting*, which is a combination of values of variables from $M$ with the property of being optimal for at least one combination of values of context variables. Depending on the algorithm, the considered set of contexts may contain all possible contexts [22] or a sample of them [7].

Given this background, $M$ is a module if the number of its context-optimal settings is smaller than the number of all possible settings of variables in $M$. It turns out that, under assumption that the modules are hierarchically organized, it is possible to effectively and robustly detect the modules without considering all possible contexts. This idea has been exploited by Watson, Thierens, and de Jong, who designed the hierarchical genetic algorithm (HGA) and have shown in [7] that it effectively solves a subclass of artificial hierarchical modular problems [6]. This result was obtained in the realm of Boolean problems, where enumerating the settings of a set of variables is possible; an extension of this approach to real-valued problems is still to come.

## 3 Functional Modularity

The aforementioned variable-based concepts of module and its context-optimal setting cannot be directly transplanted into the GP domain. First of all, variables in optimization problems lack natural counterparts in GP. Secondly, even if we identify a variable with, e.g., a specific *locus* of GP tree and treat a set of such *loci* as a module, then the interplay between such a module and the remaining part of the tree is complex and, in general, cannot be easily modelled using fitness contributions.

Thirdly, adopting such module definition still would not enable us to borrow some concepts from HGA [7], as it would be computationally too expensive to enumerate all possible settings of a module (a subtree in such case). And, last but not least, the genotype-phenotype mapping in GP is many-to-one, i.e., different GP subtrees may have the same semantics, so considering all of them seems superfluous.

However, we hypothesize that discovery of a module is possible without finding its optimal settings. Using more general terms, the need for a specific module in solution structure may be detected based on the structure (some characteristics) of the solution space alone, even if we do not know what is the best *value* (setting) for that module. To justify this claim, let us consider an illustrative optimization task of designing a battery-powered torch, with the optimized objective (fitness) being the time for which the torch'es brightness sustains a predefined threshold (maximal uptime). Let us delineate two main torch parts: the battery and the bulb. For the sake of this thought experiment, the battery will play the role of a module, while the bulb will act as the context.

The common-sense knowledge suggests that the design of the battery is (at least to some extent) independent of the design of the bulb. Some battery designs are better than others, and some of them may be considered optimal, meaning that they maximize the overall quality (fitness) of the entire solution (torch).

Let us now point out an important feature of modularity in this example. We do not need to test the fitness of the torch to assess the quality of the battery it contains. There may exist other *quality measures,* able to evaluate the settings of the module in a way that is consistent with the fitness function. For the above example, it might be the case that measuring the loss of battery voltage after it has been short-circuited for a certain time is sufficient to accurately estimate the maximal uptime of the torch. Even if such a perfectly consistent measure does not exist, then it is likely that we can find some approximate surrogate for it, which is sufficiently correlated with the fitness function. The existence of the former, fully consistent measure would imply separability, the existence of the latter, approximate estimator – modular interdependence (see Section 1 and [21]).

This example illustrates the possibility of detecting the need for a module in problem structure by discovering the proper decomposition of solution *and* finding an appropriate measure of module quality. To delineate such approach to defining and analyzing modularity in GP from the more common syntax-based (or genotype-based) methods, we coin the term *functional modularity.*

Finding a fully consistent measure of module quality may be difficult or impossible, as it subsumes separability, which is infrequent in real-world design problems and unrealistic in GP context for the reasons listed at the beginning of this section. In general, the more interdependent the module and its context are, the more complex the relation between such measures and the fitness function. In such circumstances, rather than making a qualitative *decision* about module existence, it may be better to quantify the *degree* of modularity of a particular module candidate. Analogously, it seems more reasonable to consider multiple quality measures and evaluate their utilities for module search, than pursuing the search for the ultimate best-of-all quality

measure that may never be found. These observations motivated our formal definition of the functional module introduced in following.

### 3.1 Formalization

Let $X$ be the set of all programs (solution space of the problem of consideration), and let $f : X \to \mathbb{R}$ be a maximized fitness function. A *binary solution decomposition function* (decomposition for short) is any invertible function $d : X \to P \times C$ that decomposes a solution into two components, i.e., such that:

$$\forall x \in X : d(x) = (p, c),\ d^{-1}(p, c) = x, \tag{1}$$

where $p$ is called a *$d$-part of* $x$ and $c$ is the $d$-context of $x$. In following, $p$ and $c$ are referred to as *part* and *context* and written as $p(x)$ and $c(x)$ when obtained from solution $x$. We also assume that $d$ is given and fixed, thus omitting the $d$ prefix. $P$ and $C$ are the sets of all possible parts and contexts, respectively. For the decomposition $d$ to be non-trivial, $p(x) \neq x$ and $c(x) \neq x$ must hold.

In general, elements of $X$, $P$ and $C$ are *programs*. Or, if one would like to reserve the term 'program' to the piece of code that solves the *entire* problem, then the elements of $P$ and $C$ should be called *subprograms*. The solution decomposition function $d$ must observe the constraints imposed by the syntax of the language used for representing solutions, so that the parts and contexts constitute independently executable pieces of code[1]. However, in the simple case of type-less Koza-I-style GP considered in this paper, the distinction between $X$, $P$ and $C$ is immaterial: $P$ and $C$ are equivalent to $X$, which is in this case the set of all trees that may be generated given the set of terminals and set of functions (nonterminals).

A *part quality function* is any real-valued function $f_P : P \to \mathbb{R}$. A part quality function will be identified with a *subgoal* that parts are supposed to optimize, analogously to the way solutions optimize the fitness function $f$. We assume positive preference ordering on $f_P$, i..e., we aim at maximizing its value.

By *monotonicity degree* (*monotonicity* for short) $m(f_P, f)$ of $f_P$ with respect to $f$ we mean a real-valued function that measures some form of monotonicity (strict, weak, or partial) between the values returned by $f_P$ and the values returned by $f$. In this paper, we equate monotonicity with Spearman's rank correlation coefficient. Thanks to this, we can abstract from the metric scale of fitness and focus on actual ordering of $f$ and $f_P$ values. Technically, this measure is equivalent to the Pearson's correlation coefficient with ranks substituted for $f$ and $f_P$:

$$m(f_P, f) := \rho_X(\mathbf{R}f_P, \mathbf{R}f) = \frac{1}{\sigma_{\mathbf{R}f}\sigma_{\mathbf{R}f_P}} \sum_{x\in X} (\mathbf{R}f(x) - \overline{\mathbf{R}f})(\mathbf{R}f_P(p(x)) - \overline{\mathbf{R}f_P}), \tag{2}$$

where $\mathbf{R}f_P$ and $\mathbf{R}f$ are raw scores of $f_P$ and $f$ converted to ranks. $\sigma_{\mathbf{R}f}$ and $\sigma_{\mathbf{R}f_p}$ denote the standard deviations of $\mathbf{R}f_P$ and $\mathbf{R}f$ respectively. Other reasonable definitions of monotonicity include Kendall's tau and ordinal contingency.

[1] More technically, it will become clear later that at least the part $p$ has to be independently executable.

Monotonicity measures how much the quality of the part in $f_P$ sense is correlated with the fitness function $f$ over the entire population of solutions $X$. An *optimal part quality function* $f_P^*$ is any part quality function with the highest monotonicity:

$$f_P^* = \arg \max_{f_P : P \to \mathbb{R}} m(f_P, f) \tag{3}$$

A problem given by solution space $X$ and fitness function $f$ is $\alpha$-*modular* under the assumed solution decomposition function $d : X \to P \times C$ iff

$$m(f_P^*, f) \geq \alpha. \tag{4}$$

Let us now illustrate these notions in terms of the torch example presented earlier. In that case, $X$ is the population of all possible torch designs and $f$ is the torch fitness measure as defined in our thought experiment. The solution decomposition function $d$ decomposes a torch $x \in X$ a into a battery $p = p(x)$ and a bulb $c = c(x)$, and the sets $P$ and $C$ have the interpretation of, respectively, the populations of all batteries and all bulbs that are $d$-compatible with the torch design, i.e., batteries and bulbs that may be assembled into a torch using $d^{-1}$. The $f_P$ functions are different battery quality measures: some of them are highly monotone with respect to $f$ (e.g., the initial battery voltage), some of them not (e.g., battery color). The torch design problem is $\alpha$-modular if there exist a battery (part) quality function $f_P$ that has monotonicity $\geq \alpha$.

To summarize, a problem is $\alpha$-modular if two conditions are fulfilled:

1. There exists a way of decomposing the problem into parts (solutions decomposition function $d$).
2. For the given $d$, there exists a part quality function with monotonicity $\geq \alpha$.

The rationale behind such definition of functional modularity is obviously motivated by the possible benefits from problem decomposition. Its exploitation could proceed as in the following scenario (though other approaches are conceivable). If we knew the solution decomposition function $d$ and the corresponding optimal part quality function $f_P^*$, and if its monotonicity with respect to $f$ would be sufficiently high, we could decompose the problem using $d$. Then we could use $f_P^*$ to search for the optimal part $p^*$; this search would take place in the solution space $P$, which we expect to be smaller than $X$. Finally, having found $p^*$ (or its good approximation), we could simplify the search in $X$ (e.g., make it converge faster) by considering only solutions $x$ such that $x = d^{-1}(p^*, c)$, i.e., searching only the space of contexts.

In the extreme case of $m(f_P^*, f) = 1$, such proceeding would guarantee finding the optimal solution, provided we could find $p^*$. On the other hand, this is a degenerate case that is beyond our interest: perfect correlation between part quality function $f_P$ and fitness function $f$ would imply that $f$ does not depend on context; $f_P$ would account for (explain) *all* the variability of $f$ across solutions in $X$. In such a case, context could be ignored, so no problem decomposition in the semantic sense would take place.

## 3.2 Functional modularity for case-based problems

To enable practical realization we need to constrain $f_P$ to some implementable form. As solution parts $p \in P$ are *programs*, two following classes of part quality functions seem natural: syntactic and semantic ones.

By syntactic part quality function we mean a function $f_P$ that relies exclusively on the code of program $p$, i.e., how it *looks*. Such quality functions are appropriate for, among others, problems that are decomposable due to independency between particular components of program input, feed into GP tree via terminals. A simple example could be here a bivariate symbolic regression aimed at finding the $3v_1^2 + 2v_2^2$ function: a decomposition into two univariate problems constrained to particular variables $(v_1, v_2)$ is here obvious. $f_P$ should in such a case prefer parts (program fragments) that use only some of the terminals, letting the remaining terminals to be used in the context. Such decomposition related to structure of program input data is sometimes known in advance thanks to domain knowledge; however, for many real-world this particular type of decomposability cannot be assumed.

A nice property of the functional approach to modularity is its applicability to other, non-syntactic properties of parts. We focus here on specific class of such functions, called *semantic* part quality functions, which investigate how a program *works* in order to assess its quality. In computer science, various definitions of program semantics have been proposed in past; here, following former research in GP[17, 4], we define the semantics of a program by what it does to the input *data*. We assume that a set of $l$ *fitness cases* is given that the programs (or program parts) may be applied to. For a solution $x$, its semantics $\mathbf{x}$ is the vector of actual output values it produces for the consecutive fitness cases. The particular type of the elements of $\mathbf{x}$ (Booleans, reals, images, etc.) is irrelevant for our approach.

We also assume that the desired values to be returned by the program are known for all fitness cases, and that the fitness function $f$ measures some form of similarity between $\mathbf{x}$ and the desired outputs. Formally, we redefine $f$ as $f(x) := s(\mathbf{x}, \mathbf{f})$, where $s$ is a similarity metric and $\mathbf{f}$ is the vector of desired outputs. For the class of logical problems considered in the next section, we base our maximized fitness on the Hamming distance $h$:

$$s(\mathbf{x}, \mathbf{f}) := l - h(\mathbf{x}, \mathbf{f}). \quad (5)$$

Under these assumptions there is one-to-one correspondence between fitness functions $f$ and the vectors of desired output values $\mathbf{f}$. By the same token, we assume that each part quality function $f_P$ corresponds one-to-one to a vector of desired output values $\mathbf{f}_P$, called *subgoal*, and is redefined as $f_P(p) := s(\mathbf{p}, \mathbf{f}_P)$, where $\mathbf{p}$ is the vector of actual output values produced by part $p$. For simplicity we assume that the same similarity measure is used for $\mathbf{x}$'s and $\mathbf{p}$'s, though in some cases this may not hold (e.g., typed GP trees may require different measures for $\mathbf{x}$s and $\mathbf{p}$s). Henceforth, the symbols $f$ and $\mathbf{f}$ as well as the symbols $f_P$ and $\mathbf{f}_P$ will be used interchangeably.

Let us note that in GP, it sounds reasonable to equate the above syntax–semantics distinction with the genotype–phenotype dichotomy. The authors would like to promote this viewpoint, despite the fact that opinions on what is the phenotype of a GP program vary substantially across the GP community, ranging from the one used

here (program semantic) to its identification with the genotype. However, dictionary lookup clearly favours the former case: phenotype is typically defined as 'the set of observable characteristics of an individual resulting from the interaction of its genotype with the environment' [1]. The actions carried out by a program on data may be definitely considered as genotype's interaction with the environment. Also, such interpretation of GP phenotype is consistent with the mounting agreement that phenotype should not be limited to physical manifestation in the form of an organism, but embrace the entirety of genotype's expression (cf. Dawkin's concept of extended phenotype [5]). Therefore, in our framework the syntactic quality functions work on program genotype (a tree in the case of tree-based GP), while the semantic ones – on program phenotype.

## 4 Experimental Analysis of Functional Modularity

In a long run, we are interested in exploiting the modularity for the sake of improving search convergence and other properties of the search algorithm and/or the evolved solutions. However, as mentioned earlier, the necessary precondition for modularity exploitation is modularity *detection*. Thus, in the experimental part of this study we investigate mainly the distribution of monotonicity within different GP problems and their relationships with fitness, without actually running the evolution. More technically, all the results quoted in following have been obtained by computing appropriate statistics from the same, problem- and instance-independent random sample of GP individuals.

We approach two Boolean problems, called *Xor* and *Or-And* in following, and consider four instances for both of them: 3, 4, 6, and 8-bit, where the number of bits is also the number of input variables $v_i$. The optimal solution in case of the *Xor* instances is an expression that implements the exclusive-or operation on all the input variables. In case of *Or-And* problems, the optimal solution is any expression that implements the same semantics as the following expressions:

- *Or-And-3*: (OR (AND $v_1$ $v_2$) $v_3$),
- *Or-And-4*: (OR (AND $v_1$ $v_2$) (AND $v_3$ $v_4$)),
- *Or-And-6*: (OR (OR (AND $v_1$ $v_2$) (AND $v_3$ $v_4$)) (AND $v_5$ $v_6$)),
- *Or-And-8*: (OR (OR (AND $v_1$ $v_2$) (AND $v_3$ $v_4$)) (OR (AND $v_5$ $v_6$) (AND $v_7$ $v_8$))),

For the 3-bit problems, assuming the typical ordering of the $l = 2^3 = 8$ fitness cases $(v_3, v_2, v_1) = [000, 001, \ldots, 111]$, the vector of desired values **f** is [01101001] for Xor-3 problem and [00011111] for Or-And-3. By analogy, for 4-bit problems we have $l = 2^4 = 16$ fitness cases and the vector of desired values **f** is [0110100110010110] for Xor-4 and [0001000100011111] for Or-And-4, assuming an ordering of fitness cases from 0000 to 1111. Analogous desired semantics of lengths 64 and 256 can be calculated for 6- and 8-bit problems, respectively.

Different structures of modularity are expected to emerge for these problems. We anticipate *Xor* to exhibit weaker modularity due to its perfect symmetry with respect to the independent variables (swapping the variables does not change the desired value) and due to hypothesized 'ruggedness' of its fitness landscape, resulting from extreme sensitivity to the states of single variables (flipping any input variable flips the desired output value).

Monotonicity of subgoals is estimated for each problem separately. For 3-bit and 4-bit problems, the small number of fitness cases $l = 8$ (or $l = 16$) allows us to consider all possible $2^8 = 256$ (or $2^{16} = 65536$) part quality functions (subgoals). For 6-bit and 8-bit problems, the numbers of subgoals are prohibitively large. Thus, for these instances we limit our considerations to a sample of 1,000 subgoals – one being the appropriate vector of desired values $\mathbf{f}$ and 999 randomly generated ones.

Ideally, one would estimate monotonicity by enumerating all possible solutions in $X$ (see Formula (2)) for a certain expression tree depth limit (17 in case of standard 'Koza-I-style' parameter settings). This is unfortunatelly computationally infeasible. Thus, we estimate monotonicity from a sample of 1,000,000 individuals generated using ramped half-and-half method with the ramp from 2 to 10 inclusive. Other parameters are set according to standard settings for the parity problem as implemented in the ECJ software library [16]. In particular, we use function set consisting of four binary functions: *And*, *Or*, *Nand*, and *Nor*.

Each individual from the sample is partitioned into part $p$ and context $c$ using decomposition function $d$ defined as $d(x)$ = ('the leftmost child of the root node of $x$', 'the remaining part of $x$') (see Formula (1)). For instance, the result of decomposition function applied to the tree that defines our *Or-And-3* problem is $d((OR(AND\, v_1\, v_2)\, v_3)) = (p, c) = ((AND\, v_1\, v_2), (OR \# v_3))$, where $\#$ denotes an empty tree branch. If $x$ contains only one node and thus the leftmost child of the root is absent, we ignore the individual and generate another one in its place. Next, for each subgoal, we calculate its monotonicity (see Formula (2)) using our sample of individuals.

The symmetry of the space of Boolean functions is an important feature that must not be ignored in this study. For any fitness function $\mathbf{f}$ and its logical negation $\overline{\mathbf{f}}$ it holds that $s(\mathbf{x}, \mathbf{f}) = l - s(\mathbf{x}, \overline{\mathbf{f}})$ for all $\mathbf{x} \in X$. This implies that an evolutionary process driven by fitness function $\mathbf{f}$ essentially solves *two* problems at a time: that of minimizing the individual's distance from $\mathbf{f}$ and that of maximizing its distance from $\overline{\mathbf{f}}$. Technically, solving a problem given by $\mathbf{f}$ is equivalent to solving a problem given by $\overline{\mathbf{f}}$.

Another property of Boolean problems that stems from the choice of functions is that the *a priori* probabilities of generating an individual with semantic $\mathbf{x}$ and that of generating an individual with semantic $\overline{\mathbf{x}}$ are in general very close. In particular, for the specific set of functions used in this study these probabilities are *exactly* the same. Obviously, the same can be said about the semantics of parts, $\mathbf{p}$ and $\overline{\mathbf{p}}$. As a consequence, an individual that contributes positively to monotonicity of a subgoal $\mathbf{f}_P$, necessarily contributes negatively to monotonicity of subgoal $\overline{\mathbf{f}}_P$. Over the sample, these contributions greatly compensate each other and the resulting monotonicities are very close to zero.

To compensate for this, we employ a de-symmetrized fitness function by redefining $s$ in the following way (cf. Formula (5)):

$$s(\mathbf{x}, \mathbf{f}) := l - \min(h(\mathbf{x}, \mathbf{f}),\, h(\mathbf{x}, \overline{\mathbf{f}})). \tag{6}$$

So, we consider the $\mathbf{f}$ task and the $\overline{\mathbf{f}}$ task equivalent, and an individual is rewarded either for optimizing $\mathbf{f}$ or optimizing $\overline{\mathbf{f}}$, whatever it is better at. We adopt an analogous modification to part quality functions $f_P$. As a result, $f$ and $f_P$ range from $l/2$ to $l$.

### 4.1 Intra-problem monotonicity distribution

Figures 1 and 2 present the monotonicity of subgoals for particular instances of the *Xor* problem and the *Or-And* problem, respectively. The subgoals have been ordered according to the increasing value of monotonicity. For clarity, the horizontal axes are labelled only by the first and the last subgoal in this order. Let us remind that for 3- and 4-bit instances, these axes list *all* possible subgoals, while for the 6- and 8-bit instances, they embrace the 1000 subgoals from the sample. For the former instances, the abscissa labels illustrate the semantics of the lowest- and the highest-ranked subgoals; for the latter instances, semantics are too long and hence not shown.

It is easy to notice that for both problems and all instances, monotonicity significantly varies across subgoals. In particular, highly-monotonous subgoals are infrequent. Also, some subgoals are 'deceptive' in the sense that they have remarkably negative monotonicity. This result indicates that, in the space of semantics of parts (subgoals), there are points ('good' subgoals) with the property that if the semantic of the part becomes more similar to one of them, then the fitness of the entire individual/solution is likely to increase. And conversely, there are also 'bad' subgoals with the inverse property. In short: monotonicity differentiates the subgoals in a way that is consistent with the fitness landscape.

In terms of $\alpha$-monotonicity (Formula (4)), *Xor-3* is approximately 0.35-modular as the maximum monotonicity over all subgoals amounts here to 0.3497. *Or-And-3* has the maximum subgoal monotonicity around 0.4 (precisely: 0.4005).

An analogous relation holds for 4-bit instances: *Xor-4* is 0.41-modular and *Or-And-4* is 0.46-modular. For these instances, the ranges of monotonicity are wider than for 3-bit: $[-0.18, 0.41]$ for *Xor-4*, and $[-0.40, 0.46]$ for *Or-And-4*. It may be observed that the analogous ranges for 6- and 8-bit instances of *Xor* problem are wider than for smaller ones, but for the *Or-And* problem the tendency is completely opposite – the range of monotonicity for *Or-And-8* instance is much narrower than for *Or-And-3*.

### 4.2 Inter-problem differences in monotonicity

The mean monotonicity over all subgoals is very close to zero for both problems and all instances. However, the graphs in Figures 1 and 2 clearly demonstrate that *Or-And* and *Xor* have notably different distributions of monotonicity across the subgoals: for

instance, the monotonicity of *Or-And-3* subgoals seems more dispersed than of *Xor-3*. This is confirmed by standard deviations, which amount to 0.0973 and 0.1191 for *Xor-3* and *Or-And-3*, respectively. The difference between the corresponding variances seems statistically significant ($p$-value of $F$-test $< 0.05$), though the variables are not normally distributed. For the 4-bit problems, the difference is also remarkable (0.0705 for *Xor-4* and 0.1641 for *Or-And-4*) and significant ($p < 10^{-30}$). This indicates that the *Or-And* problems are more modular than the *Xor* problems. Interestingly, this observation seems to hold independently on instance size. Thus, different problems tend to exhibit different structure of monotonicity.

Not surprisingly, the subgoal that maximizes monotonicity is that one which is equivalent to the vector $\mathbf{f}$ of desired values of the fitness function. This holds for all problems and instances; e.g., for *And-Or-3*, that subgoal is 00011111. Explanation of this observation is straightforward: if by chance solution's part $p(x)$ returns $\mathbf{f}$ and the processing performed by the context does not affect it, so that the entire tree returns $\mathbf{f}$, the solution's contribution to subgoal's monotonicity is very high.

Because of this phenomenon, we observe a prominent peak of monotonicity at the rightmost end of all charts. The monotonicities of subsequently ranked subgoals are remarkably lower. However, this decline is more prominent for the *Xor* problems, particularly for 6- and 8-bit instances; for *Or-And*, the group of subgoals having monotonicity around 0.2 is quite numerous, while for *Xor* it is empty for most instances. This may indicate that the *Or-And* problem is more modular and gives more hope for being automatically decomposed.

### 4.3 Relation between monotonicity and fitness

The previous two sections focused on the global, unconditional, distribution of monotonicity in the sample of subgoals. The objective of the analysis presented in following is to relate those results to the fitness of complete solutions. More technically, we verify whether the well-performing solutions are more modular than the other ones.

We do that by analyzing in more detail the sample used for the estimation in Sections 4.1 and 4.2. To this aim, we group the 1,000,000 individuals with respect to fitness. Then, within each fitness group, we perform the following: from each individual $x$, we extract its part $p(x)$, calculate its semantics $\mathbf{p}(x)$, and find the subgoal that is most similar to it in terms of Hamming distance (including the desymmetrization process). This results in each individual in each fitness group having one subgoal assigned to it. Finally, we average the monotonicities within the fitness group, obtaining the mean intra-group monotonicity.

When a fitness group is smaller than five individuals, we restrain from calculating the mean, as this would imply very wide confidence intervals and would not tell us much. This happened in all 4-or-more-bits instances except for *Or-And-4* and was obviously more likely for higher fitness values. For *Xor-4* the best reached fitness of 14 was attained by only one out of 1,000,000 individuals in the sample, for *Xor-6* only two individuals scored 39 (and none more), and for *Xor-8* only four individuals scored

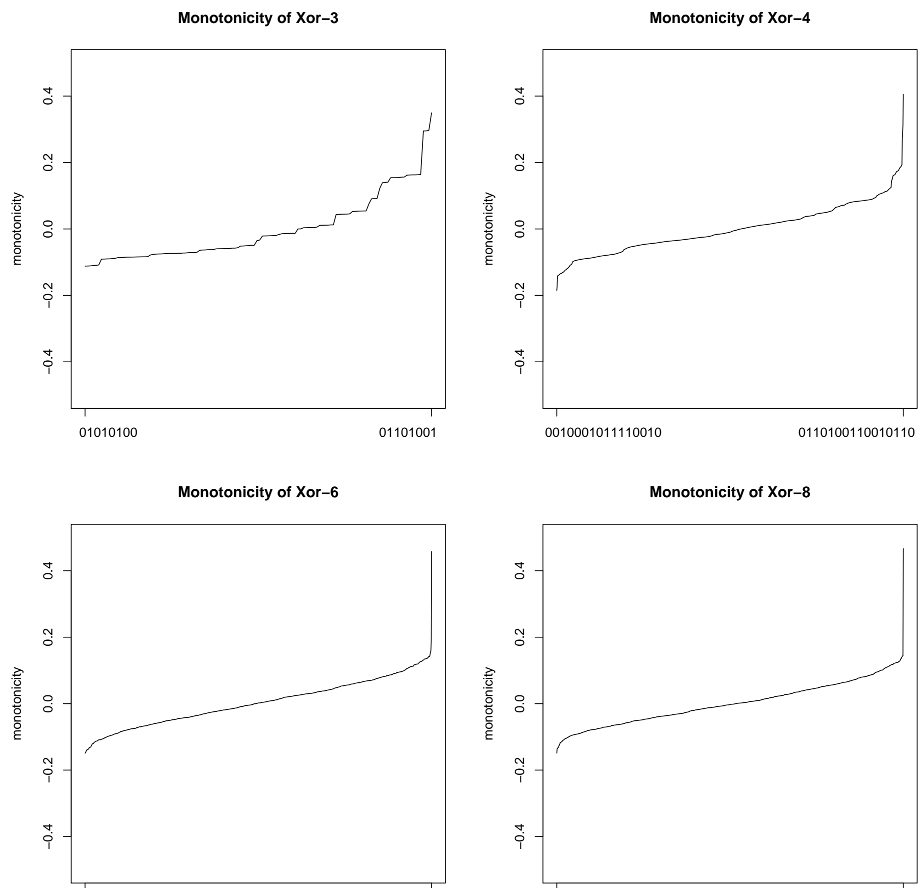


**Figure 1**: Monotonicity of all subgoals for Xor problem estimated from 1,000,000 random individuals.

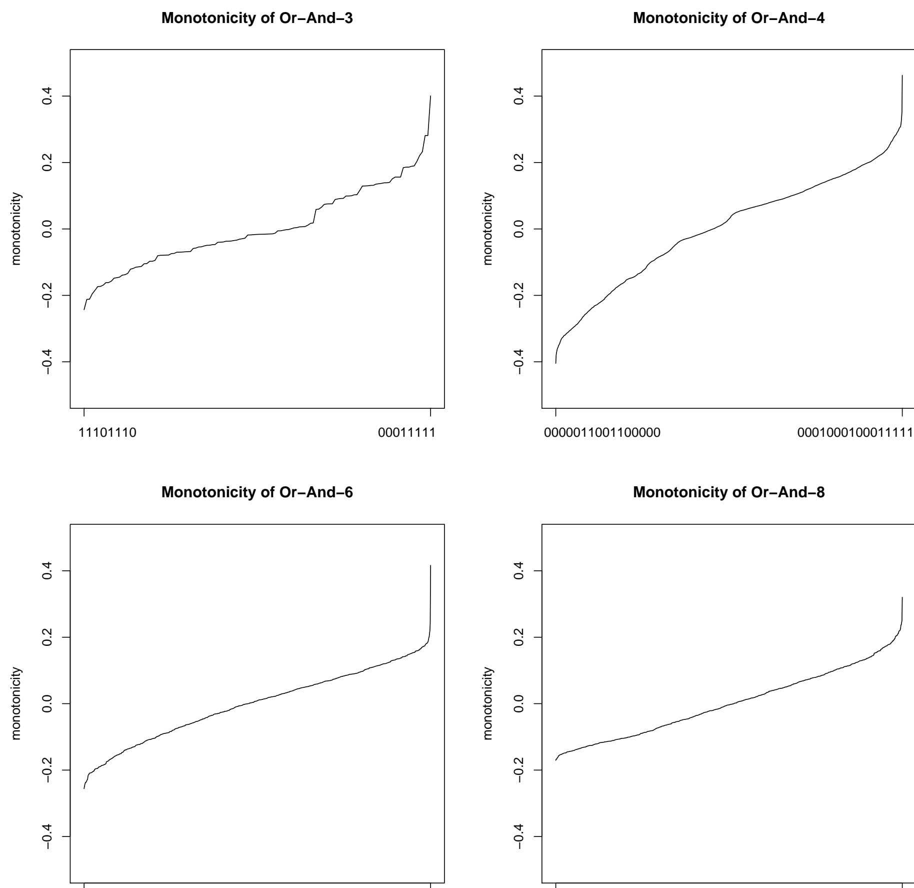


**Figure 2**: Monotonicity of all subgoals for Or-And problem estimated from 1,000,000 random individuals.

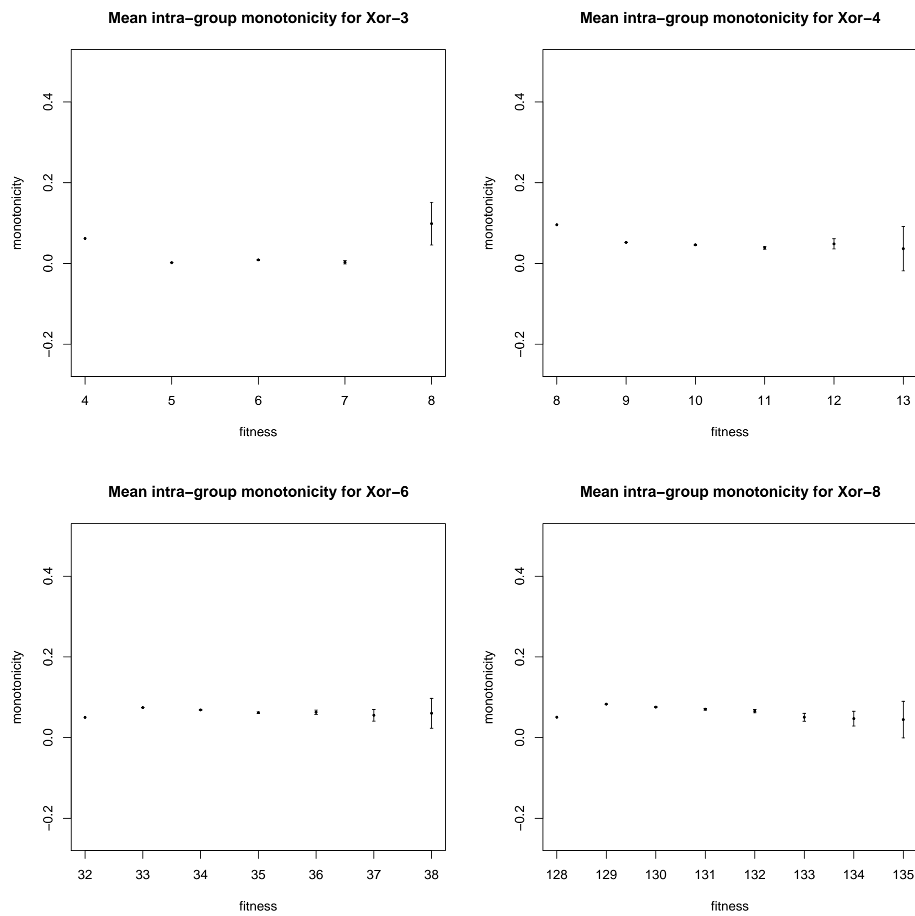


**Figure 3**: Mean intra-group monotonicity of Xor instances estimated from 1,000,000 random individuals.

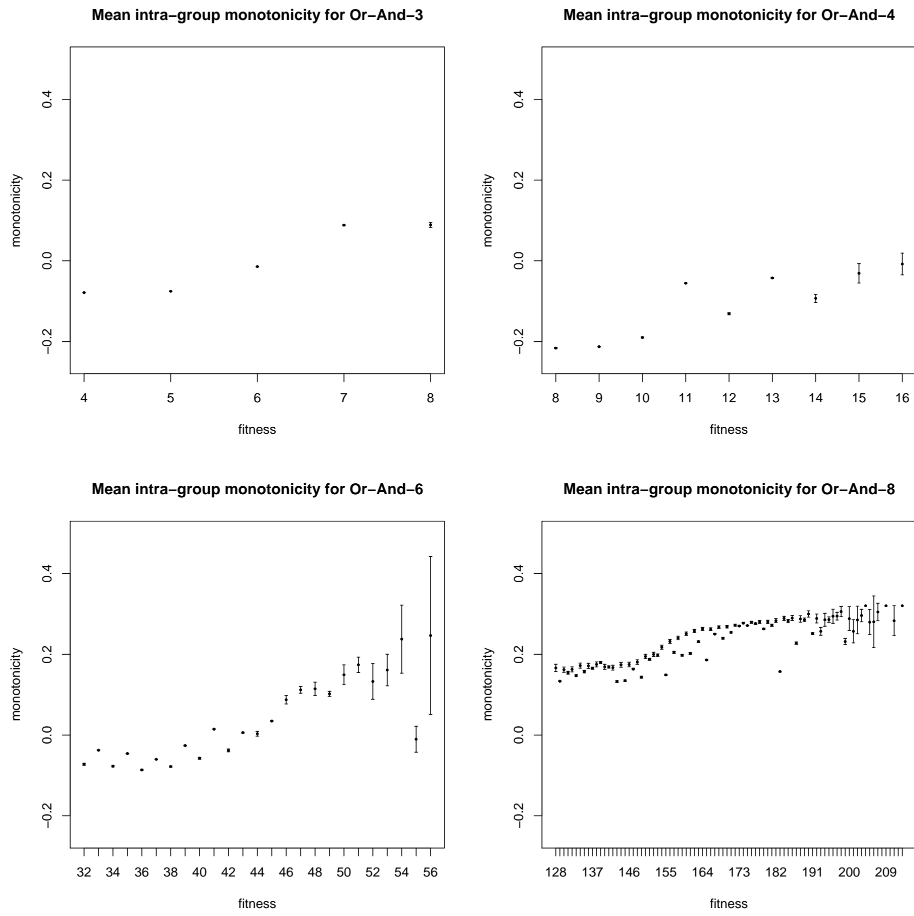


**Figure 4**: Mean intra-group monotonicity of Or-And instances estimated from 1,000,000 random individuals.

better than 135. As the *Or-And* problem is much easier, for 3- and 4-bits instances each possible fitness was attained at least by several hundreds random individuals. For bigger instances of *Or-And*, similarly to *Xor*, many fitness groups did not contain at least five individuals and are not shown in the graph.

Let us emphasize that matching part semantics $\mathbf{p}(x)$ to subgoals is qualitatively different depending on instance size. For 3- and 4-bit instances, the set of considered subgoals is complete, so finding a subgoal that perfectly matches part semantics (in other words, the subgoal that is *realized* by the part $p(x)$; zero Hamming distance) is guaranteed. This is not the case for 6- and 8-bit instances where the subgoal sample is only a tiny fraction of entire subgoal space: here, the Hamming-closest subgoal may differ from $\mathbf{p}(x)$ on an arbitrary number of bits (fitness cases).

Figures 3 and 4 present the values of the mean intra-group monotonicity for particular fitness groups accompanied by 0.95-confidence intervals. For the above-mentioned reason, the horizontal axes do not reach the optimum fitness values for most instances. For the difficult *Xor* problems, the means are rather close to zero, similar to each other, and do not seem to change systematically; for larger instances, they even slightly decrease with fitness. This suggests that, in terms of fitness, there are no definitely good and definitely bad subgoals in this problem. However, for the presumably more modular (with respect to assumed decomposition function $d$) *Or-And* problem, the values of intra-group monotonicity are more diversified and seem to increase with fitness. This demonstrates that for *Or-And*, monotonicity can serve as an indicator of subgoal's usefulness. The fact that for the *Or-And-4* instance all the means are negative does not undermine this conclusion. As mentioned in Section 4.2, the overall average monotonicity for a given problem is virtually zero, so what counts are the *differences* between fitness groups.

For the larger instances of *Or-And*, a remarkable bifurcation of the monotonicity can be observed that follows the pattern of alternating fitness values: mean monotonicity for odd-fitness groups is often significantly higher than the means in the adjacent even-fitness groups. This artefact inclined us to investigate the parity of semantics in our sample. As it turned out, the Boolean functions used in our setup together with all input variables (GP terminals) having an even number of ones in their semantics introduce a strong bias in favor of trees that have semantics with an even number of ones. The odds for that increase with instance size and varying from around 3:1 (for the sample used for 3-bit instances) to 9:1 (for the sample used for 8-bit instances). This obviously affects also parts, but because they are smaller, the odds are even more extreme: from 4.5:1 to 12:1, respectively. We hypothesize that this introduces some form of imbalance in sampling the semantic spaces of solutions and trees, which in turns results in the observed bifurcation.

## 5 Discussion and Conclusions

The experimental results authorize us to formulate the following claims:

1. For some problem instances, different subgoals tend to have significantly different monotonicity.

2. Problem instances display different structure of monotonicity, meant as the characteristics of the distribution of monotonicity across the subgoals.

3. For some problem instances, the monotonicity of solution's closest subgoal positively correlates with its fitness.

To some extent, these observations may be generalized from problem instances to problems (*Xor* and *Or-And*). Because there is no reason to suppose that for all other problems of logic function synthesis it should be different, we can hypothesize that some problems tend to be less modular (close-to-uniform distribution of monotonicity), and some more modular (significantly non-uniform distribution of monotonicity). This gives hope for delineating the class of semantically modular problems. For such problems, decomposition based on functional modularity is likely to provide better scalability and enable solving problems that remain intractable using contemporary computational resources.

Many questions and research issues pertaining to this study remain open. To become applicable in practice, functional modularity has to be extended beyond the realm of Boolean problems, which may prove difficult due to non-discrete form of semantics. Also, as requiring a human to provide the appropriate decomposition function $d$ is far from realistic in most real-world scenarios, a complete decomposition algorithm should be able to discover it autonomously. In particular, for the sake of clarity, in this paper we used a specific decomposition function $d$ that defines part as the left-hand subtree of the root node. It should be emphasized that this form of decomposition has potential drawbacks: the context is forced to aggregate the output produced by the part with the values returned by the context using a *single* operation. The existence of ideal solutions for the problems considered in this paper implies that such aggregation is possible, yet not necessarily for *all* possible subgoals. Moreover, even if for some subgoal there exists an optimal context that makes the entire solution ideal, then from the viewpoint of search effort of an evolutionary run (measured, e.g., as the expected number of evaluations needed to find the optimum), some subgoals and some contexts may be easier to optimize than others.

We hypothesize that using other decomposition functions $d$ can alleviate this difficulty. A simple example of such function could be $d$ that defines part as the left-hand child of left-hand child of the root, and context as the remaining part of the tree. In such a case, context would have *two* operations at its disposal to combine the output of the part with the values computed by the context, and it could use different right-hand arguments for these operations. Therefore, there would be more 'degrees of freedom' in the search process and it should be easier to evolve an optimal context.

It is interesting to note that this line of reasoning leads in the end to decomposition functions $d$ that define part a the leftmost *leaf* of the tree. Such decomposition

definitely gives a lot to say to the context, so that the abovementioned risk of the context not being able to incorporate the semantics of the part is neglectable. However, a part defined in such a way is very unlikely to have any impact on the overall semantics of the entire tree. Thus, we conclude that using different decomposition functions allows us to control the trade-off between the difficulty of optimizing the context and the contribution of part semantics to the entire solution.

Another question pertains to computational efficiency: in terms of the expected time required to find the optimal solution, does it pay off to decompose the problem into two subproblems and solve them independently, given the extra overhead imposed by the analysis of monotonicity? And if yes, then when? Finally, a more complete theory supporting this approach would be of much help.

The above experimental analysis is a proof-of-concept demonstrating that functional modularity may be helpful for characterizing the compositionality and difficulty of a problem. Knowing the structure of modularity for a particular problem is the first step for effective exploitation of monotonicity, which we will pursue in future research.

## Acknowledgments

This research has been supported by research grants DS-91-471 and N N519 3505 33.

## References

[1] *New Oxford American Dictionary.* 2008.

[2] P. J. Angeline and J. B. Pollack. The evolutionary induction of subroutines. In *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society*, pages 236–241, Bloomington, Indiana, USA, 1992. Lawrence Erlbaum.

[3] W. Banzhaf, D. Banscherus, and P. Dittrich. Hierarchical genetic programming using local modules. Technical Report 50/98, University of Dortmund, Dortmund, Germany, 1998.

[4] L. Beadle and C. Johnson. Semantically driven crossover in genetic programming. In J. Wang, editor, *Proceedings of the IEEE World Congress on Computational Intelligence*, pages 111–116, Hong Kong, 1-6 June 2008. IEEE Computational Intelligence Society, IEEE Press.

[5] R. Dawkins. *The Extended Phenotype : The Long Reach of the Gene.* Oxford University Press, USA, August 1999.

[6] E. D. de Jong, R. A. Watson, and D. Thierens. A generator for hierarchical problems. In *GECCO '05: Proceedings of the 2005 workshops on Genetic and evolutionary computation*, pages 321–326, New York, NY, USA, 2005. ACM.

[7] E. D. de Jong, R. A. Watson, and D. Thierens. On the complexity of hierarchical problem solving. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1201–1208, New York, NY, USA, 2005. ACM.

[8] D. Goldberg. *Genetic algorithms in search, optimization and machine learning.* Addison-Wesley, Reading, 1989.

[9] D. E. Goldberg. *The Design of Innovation: Lessons from and for Competent Genetic Algorithms.* Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[10] G. Harik. *Learning Gene Linkage to Efficiently Solve Problems of Bounded Difficulty Using Genetic Algorithms.* PhD thesis, University of Illinois at Urbana-Champaign, 1997.

[11] J. Holland. *Adaptation in natural and artificial systems*, volume 1. University of Michigan Press, Ann Arbor, 1975.

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

[13] J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press, Cambridge Massachusetts, May 1994.

[14] K. Krawiec and B. Wieloch. Functional modularity for genetic programming. In G. Raidl, F. Rothlauf, G. Squillero, R. Drechsler, T. Stuetzle, M. Birattari, C. B. Congdon, M. Middendorf, C. Blum, C. Cotta, P. Bosman, J. Grahl, J. Knowles, D. Corne, H.-G. Beyer, K. Stanley, J. F. Miller, J. van Hemert, T. Lenaerts, M. Ebner, J. Bacardit, M. O'Neill, M. Di Penta, B. Doerr, T. Jansen, R. Poli, and E. Alba, editors, *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 995–1002, Montreal, 8-12 July 2009. ACM.

[15] M. Looks. *Competent Program Evolution.* Doctor of science, Washington University, St. Louis, USA, 11 Dec. 2006.

[16] S. Luke. ECJ evolutionary computation system, 2002. (http://cs.gmu.edu/eclab/projects/ecj/).

[17] N. F. McPhee, B. Ohs, and T. Hutchison. Semantic building blocks in genetic programming. In M. O'Neill, L. Vanneschi, S. Gustafson, A. I. Esparcia Alcazar, I. De Falco, A. Della Cioppa, and E. Tarantino, editors, *Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008*, volume 4971 of *Lecture Notes in Computer Science*, pages 134–145, Naples, 26-28 Mar. 2008. Springer.

[18] Z. Michalewicz. *Genetic algorithms + data structures = evolution programs.* Springer-Verlag, Berlin, 1994.

[19] J. P. Rosca and D. H. Ballard. Genetic programming with adaptive representations. Technical Report TR 489, University of Rochester, Computer Science Department, Rochester, NY, USA, Feb. 1994.

[20] J. A. Walker and J. F. Miller. The automatic acquisition, evolution and reuse of modules in cartesian genetic programming. *IEEE Transactions on Evolutionary Computation*, 12(4):397–417, Aug. 2008.

[21] R. Watson. *Compositional Evolution: Interdisciplinary Investigations in Evolvability, Modularity, and Symbiosis.* PhD thesis, Brandeis University, 2002.

[22] R. A. Watson and J. B. Pollack. Modular interdependency in complex dynamical systems. *Artif. Life*, 11(4):445–458, 2005.

[23] D. Wolpert and W. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.