On Relationships between Semantic Diversity, Complexity and Modularity of Programming Tasks

Krzysztof Krawiec
Institute of Computing Science, Poznan University of Technology
Piotrowo 2, 60965 Poznań, Poland
krawiec@cs.put.poznan.pl

ABSTRACT

We investigate semantic properties of linear programs, both internally, by analyzing the memory states they produce during execution, and externally, by inspecting program outcomes. The main concept of the formalism we propose is program trace, which reflects the behavior of program in semantic space. It allows us to characterize programming tasks in terms of traces of programs that solve them, and to propose certain measures that reveal their properties. We are primarily interested in measures that quantitatively characterize functional (semantic, behavioral) modularity of programming tasks. The experiments conducted on large samples of linear programs written in Push demonstrate that semantic structure varies from task to task, and reveal patterns of different forms of modularity. In particular, we identify interesting relationships between task modularity, task complexity, and program length, and conclude that a great share of programming tasks are modular.

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Heuristic methods*

Keywords

genetic programming, modularity, program semantics, Push

1. INTRODUCTION

Modularity, ubiquitous in nature and human design, is quite an elusive concept that can be interpreted in multiple ways. It is typically defined in the framework of complex systems, i.e., systems comprising multiple interconnected components. Such a system is termed modular if some of its components are interconnected stronger than others, either in a structural [2] or functional [1, 16] sense. A different concept known under this term, the propensity of solutions to be composed of repeated parts (cf. body segments in animals), is outside the scope of this paper.

In genetic programming (GP), it is common to characterize modularity in direct relation to program code, and identify modules with code fragments (typically subprograms). In this perspective, a module is a subprogram that turns out to be beneficial for individuals' fitness. This framing forms the foundation for automatically defined functions and various methods of code encapsulation (see, e.g., [11, 15]).

This perspective on modularity in GP has however certain limitations. In contrast to representations in which modules naturally reveal themselves in solution's structure, e.g., in connections in neural and logical networks [2], modules in GP cannot be usually detected by means of a purely structural analysis. For most program representations, particularly for trees, one cannot tell if a subprogram constitutes a module in certain task, by looking at a single solution to that task. To find out whether a piece of code forms a module or not, it has to be put into different contexts, and its effects on individuals' fitness have to be investigated (e.g., [6, 14]). Such an approach is however disputable, because a module cannot be expected to serve its purpose if the context (the remaining program part) is not ready to cooperate with it.

This problem can be illustrated from another perspective. Consider the programming task of writing a function that calculates a median of a list of numbers. Clearly, it can be split into two subtasks: the list should be first sorted, and then the central element has to be located. However, at least for the former subtask, there exist many possible solutions (sorting functions), which can have little in common in syntactic terms. A single program that solves this task can be useless for identifying modules.

This clearly shows that module in GP should be considered more as a *functional* entity that is responsible for certain aspect of the whole programming task. In this framing, a module exists in abstraction from specific programs, and modularity is an inherent property of a task, rather than of a single solution, a more phenotypic (behavioral, semantic) than genotypic (structural, syntactic) phenomenon. This perspective is also prevailing in general studies on modularity, not specific for GP. For instance, the NK-landscapes [3] and HIFF [16] benchmarks implement modularity as a property of task (fitness function, to be precise).

Though functional framing of modularity deserves interest, it has attracted only limited attention in GP [7, 4, 10, 9]. The possible reason for this state of matter is that the interactions between functional modules in GP are much more complex than in optimization problems. For instance,

GECCO'12, July 7-11, 2012, Philadelphia, Pennsylvania, USA.

in NK-landscapes and HIFF, modularity is an effect of different combinations of variable values bringing various contributions to fitness. In GP, one cannot expect that subprograms, which are not simply variables but entities that perform processing of external data, would contribute to program's fitness in an equally straightforward way (not mentioning the fact that equalling variables with subprograms is highly questionable in the first place). Epistasis between components is here much more sophisticated and thus more difficult to detect and model.

In this study, we attempt to develop the functional perspective on modularity in GP by investigating the relationships between program semantics and modularity. The paper has two major contributions. Firstly, we propose a unified formalism for semantic characterization of the process of program execution, program outcome, and programming task. Secondly, we provide ways to quantitatively assess modularity of programming tasks, and discover interesting relationships of these measurements to other properties, semantic diversity and program complexity.

2. PROGRAMS, TRACES AND TASKS

The core concept of GP and other approaches to automatic programming is programming task (*task* for short), which can be formulated as 'find a program that, given input, produces specific output, i.e., exhibits the desired behavior'. A task intended to be solved by humans will typically express the desired output in terms of constraints. For instance, given an input comprising three numbers x_1, x_2, x_3, the programming task of finding a sorting program could be specified as constraint $x_1 \leq x_2 \leq x_3$. In GP, one typically describes the expected behavior in a different way, more characteristic for supervised machine learning from examples: a set of examples (fitness cases) of the input-output mapping is given, and the program sought for has to replicate (or approximate) the mapping for these examples.

Both formulations try to grasp the concept of *program semantics*. The latter, by specifying the behavior only for selected inputs, can be viewed a an approximation of the former. In following we unify them into a common formal framework.

We define *program* as a finite sequence of instructions from a given set I, written in the programming language of consideration. A program interpreter, capable of executing such programs, is equipped with memory that is filled with program input prior to execution and stores program output after it terminates. All information about the intermediate states of program execution, except for the program itself and instruction pointer, is reflected by the state of memory. Let M denote a finite set of all possible memory states, and let s_0 be the vector composed of all elements from M ordered in an arbitrary way. Thus, s_0 represents also all possible inputs to any program that can be executed by the interpreter.

For a given program p, consider applying it independently to all inputs from s_0. For a start, assume that we execute the first instruction of p. For each element of s_0, this will result in a certain memory state from M. These memory states can be gathered in a vector s_1, with its elements corresponding to the elements of s_0. Consequently, execution of each subsequent instruction of p will result in analogous vectors $s_2, s_3, \ldots$. This process can be alternatively viewed as having $|M|$ interpreters, and running p in parallel on all

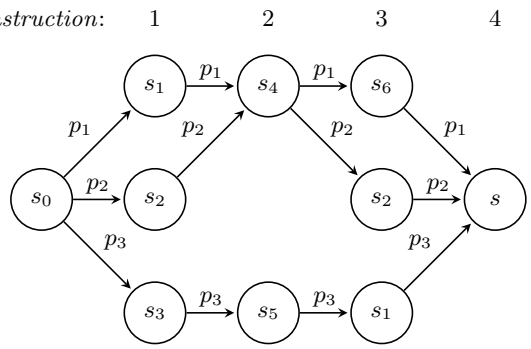

Figure 1: Graphical representation of three distinct traces of three programs p_1, p_2, p_3, each of length 4, that solve task s. Vertical columns of states correspond to consecutive instructions executed by programs. Note that a program can revisit the same state (p_2 visits state s_2 twice), and different programs can visit the same state on different stages of their execution (both p_1 and p_3 visit s_1).

of them, for all elements of s_0. Throughout this paper, program execution will be meant in this very sense.

We call the vectors $s_0, s_1, s_2, \ldots$ *states* to clearly tell them apart from the elements of M. The set of states that can be visited while executing any program in this way is a (proper or not) subset of $M^{|M|}$. The sequence of states visited when executing the consecutive instructions of program p constitutes its *trace* $t(p)$. A state can appear in a trace more than once, and different programs can have the same trace. The initial visiting of s_0 is not included in traces.

The length of a trace can be greater than the length of its program if I includes jumps or loops. In particular, non-stopping programs have infinitely long traces. For this reason, we consider programming languages that have no loop nor jump instructions, which implies that all programs halt and that the lengths of a program and its trace are equal, i.e., $|p| = |t(p)|$. When considering sets of programs, we will also assume they all have the same length l.

Let $t^i(p) \in M^{|M|}$ denote the ith state in the trace $t(p)$. The state reached at the end of p's execution, i.e., $t^l(p)$, will be referred to as *semantic* of p, as it reflects the output of p for *all* possible inputs (embraced by s_0), thus nothing more can be said about its behavior.

A *programming task* can be now formulated as follows: given $s \in M^{|M|}$, find a program p such that $t^l(p) = s$. Therefore, a task is equivalent to a specific point $s \in M^{|M|}$ that determines the desired output for all possible memory states. We say that a p such that $t^l(p) = s$ *solves* task s and that p is a *solution* to s. Let $P(s)$ denote the set of all solutions of s. A programming task is *solvable* if $P(s) \neq \emptyset$.

The set of traces of all programs that solve s will be called *task trace* and denoted by $T(s) = \{t(p) : t^l(p) = s\}$. In general, $|T(s)| \leq |P(s)|$.

Figure 1 shows graphical representation of three traces of three programs of length $l = 4$ that solve the same task s, superimposed on each other. Only the states that belong to the considered traces are shown. The illustration emphasizes the sequential nature of programs, which are executed from left to right (note the instruction numbers at the top). Note that a state can appear multiple times at different positions

in single or multiple traces. Thus, this representation is not equivalent to graph or automaton.

Trace is a convenient concept in that it embraces *all* stages of program's 'lifecycle': definition of input data, program execution, and program output. A state is a formal object capable of representing programming tasks, a semantics of a program, but also partial results of program execution. The set $M^{|M|}$ embraces the Cartesian product of all memory states that can be produced by any program given any input. Every state from $M^{|M|}$ defines a programming task (solvable or not), and *some* states are semantics of actual programs. The starting point for traces of all programs is the same (s_0), so a trace of a program is not conditioned by its input. As a set of fitness cases typically used in GP is a subset of s_0, this unifies the two views of semantics discussed earlier.

2.1 Experiment 1: Diversity and complexity

The objective of this experiment is to analyze programming tasks in a chosen programming language using the introduced formalism. We chose Push [13] and interpreter PshGP written by Jon Klein, `http://spiderland.org/Psh/`. In general, Push programs are lists of instructions that can be nested, but we disable nesting to consider only strictly linear programs of length l.

Push is a stack-based language, and its interpreter is equipped with separate stacks for different data types. We assume that computation takes place in the integer domain, and that memory state is fully determined by the state of integer stack. Two memory states are considered identical if they comprise stacks of the same depth, with the same value at each position. The input to a program is the initial stack state, and the output is the state it leaves after completion.

Our programming language comprises 8 instructions $I =$ {*, +, -, /, *neg*, *dup*, *pop*, *swap*}, all of which affect only the state of integer stack. The arithmetic operations pop two arguments from the stack and push the result on top of it; *neg* pops a number, and pushes a negative of it; *dup* pushes a copy of the topmost element; *pop* discards the topmost element; *swap* reverses the order of the two topmost elements. An instruction has no effect if the stack is too shallow. Note that, as all numbers in the stack contribute to its state, and this applies also to output stack states, the programs considered here perform in fact *multiple* regression.

We cannot consider all possible program inputs, as stacks can have an arbitrary depth. Thus, we limit the set of considered inputs to a sample of 10 integers $\{-5, -4, \ldots, 3, 4\}$. An initial memory state consists of two copies of a number placed on the stack (e.g., $(-5, -5)$ or $(3, 3)$).[1] Thus, the initial state s_0 (cf. Section 2) is a vector of 10 stacks (fitness cases), each holding two identical integers: $s_0 = [(-5, -5), \ldots, (4, 4)]$. Any other state is a vector of 10 stack states corresponding to these initial stack states.

Sampling procedure. To gain insight into the semantic characteristic of entire program space, considering a specific programming task or even a few benchmarks would be insufficient. Rather than that, we aim at characterizing a large sample of programming tasks and programs that solve them. We cannot however create tasks arbitrarily, i.e., by generating desired output state via filling the stack with random integers, as such tasks are not guaranteed to be solvable (in particular, under length limit l). Thus, we do the reverse: we generate programs and check which tasks they solve.

[1]With only one copy of the input number, a program, to produce a nonempty stack, would have to start with *dup* to provide the subsequent instructions with enough arguments. This would unnecessarily constrain the program space.

Algorithm 1 The sampling algorithm: I - instruction set, l - program length, n_t - number of unique traces to be generated, n_p - maximal number of iterations.

```
1: procedure SAMPLE(I, l, n_t, n_p)
2:     P ← ∅                          ▷ Set of unique programs generated so far
3:     T ← ∅                          ▷ Set of unique traces generated so far
4:     S ← ∅      ▷ Set of unique semantics (tasks) generated so far
5:     repeat
6:         repeat
7:             p ← RANDOMPROGRAM(I, l)
8:         until p ∉ P
9:         t ← t(p)                   ▷ The trace of the drawn program
10:        if ¬ISTRIVIAL(t) then
11:            P ← P ∪ {p}                          ▷ Update the samples
12:            T ← T ∪ {t}
13:            S ← S ∪ {t^l}
14:        end if
15:        n_p ← n_p − 1
16:    until n_p = 0 or |T| = n_t
17:    return (P, T, S)
18: end procedure
```

Systematic enumeration of $|I|^l = 8^l$ programs is infeasible even for small l, so we rely on sampling procedure SAMPLE detailed in Algorithm 1. SAMPLE produces a threefold result: a sample $\mathcal{P}$ of programs, the set $\mathcal{T}$ of program traces of programs in $\mathcal{P}$, and the set $\mathcal{S}$ of semantics of those programs, which is simply the set of final states $\{t^l(p)\}$ of traces from $\mathcal{T}$. $\mathcal{S}$ is also the set of tasks solved by the programs from $\mathcal{P}$.

In each iteration, SAMPLE generates a random program by drawing l instructions with equal probability (iid). A program has to pass several checks to get into the final sample. Firstly, it has to be unique – duplicates are rejected. Secondly, it has to be nontrivial, i.e., it cannot end with the same memory state for all fitness cases, nor produce an empty stack for one or more fitness cases. Programs that are trivial in this sense are symptomatic for stack-based languages: once the stack becomes empty, it cannot be refilled and remains empty till the end of program execution. The longer a program, the more likely it is to happen; for $l = 20$, over 95% of randomly drawn programs turn out to be trivial.

Let us also note that lines 12 and 13 of SAMPLE are frequently ineffective, as multiple programs can map onto the same trace, and multiple traces can end with the same state.

SAMPLE terminates when n_t unique traces have been collected or the maximum number of iterations n_p elapsed (total number of generated programs, whether trivial or not). These stopping conditions and the sets produced by SAMPLE will be used selectively in the subsequent experiments.

Semantic diversity of programs. We start with investigating *semantic diversity*, by which we mean the number of unique semantics of generated programs ($|\mathcal{S}|$), or, equivalently, the number of tasks solved by the sample programs. Figure 2 presents, for different program lengths, the parametric curves reflecting progression of the number of nontrivial programs ($|\mathcal{P}|$, horizontal axis) and semantic diversity ($|\mathcal{S}|$, vertical axis), with the number of generated programs (n_p). Datapoints have been obtained by calling SAMPLE for consecutive values of n_p. All curves start in the lower left corner (an empty sample, $n_p = 0$), and the end of each curve marks the point for $n_p = 50,000,000$.

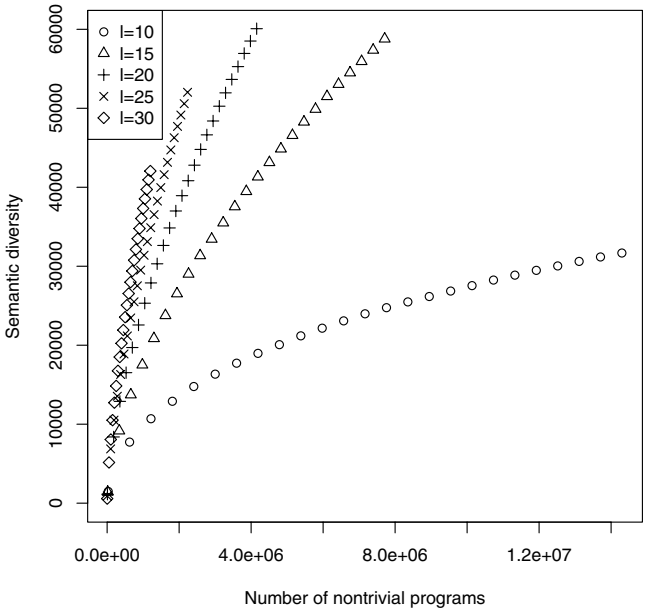

Figure 2: Number of nontrivial programs and semantic diversity as a function of the number of generated programs n_p, for different program lengths l.

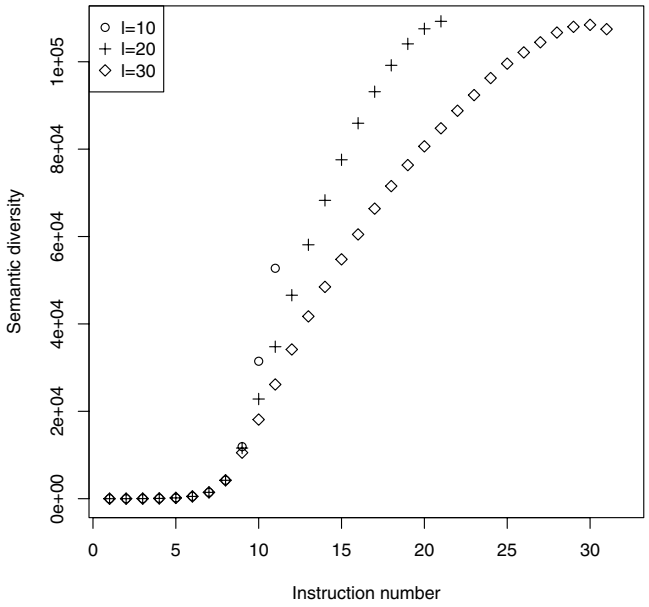

Figure 3: Internal semantic diversity of randomly generated nontrivial programs as a function of instruction number.

As program length grows, the end points of curves shift to the left, because longer programs are more likely to be trivial. Most instructions (arithmetic and *pop*) reduce stack size, so, as l increases, it becomes more probable for the stack to become empty at some point of program execution. For this reason, for maximal n_p, the number of nontrivial programs generated for $l = 30$ is almost ten times smaller than for $l = 10$, despite drawing the same number of programs.

The next effect clearly visible in Fig. 2 is that the longer the programs, the steeper the curves. If a newly generated program is nontrivial, then it is more likely to increase the semantic diversity for large l than for small l. This was expected, as, for the assumed instruction set (and for all 'reasonable' instruction sets), making programs longer cannot decrease the number of outputs they can produce. When the trivial programs are left aside, semantic diversity is a non-decreasing function of program length.

However, the distribution of program semantics is highly non-uniform, i.e., some semantics are more likely to be generated than others (cf. demonstration for tree-based GP in [5], ch. 7). For this reason, the slope of the curves decreases and the consecutive data points tend to get closer to each other. The richer the pool of already generated semantics, the harder it is to generate a novel one. The number of generated semantics approaches asymptotically the actual number of semantics that could be generated by enumeration of all programs. This is clearly visible for $l = 10$, and we expect it to become prominent for larger l, if it was not for limited computational resources.

As a net effect of the above factors, semantic diversity obtained for the largest n_p (endpoints of curves) increases with l only to a certain point. Starting from l around $15 \ldots 20$, it starts to drop, so that for $l = 25$ and 30 it is significantly smaller than for $l = 20$. There is an opposition between programs' potential to produce novel semantics, which grows with l, and the probability of generating semantically nontrivial programs, which decreases with l. An interesting consequence of this is the existence of a break-even point, which represents the optimal program length (here: $\in [15, 20]$) that allows attaining maximum semantic diversity for an assumed sample size n. This can be practically leveraged in GP, e.g., in population initialization methods.

This confirms the earlier result presented in [5] (e.g., Fig. 7.2) for program trees: with increasing l, the distribution of semantics converges to a limit. This distribution, not shown here for brevity, is very non-uniform in our case: its median is three orders of magnitude larger than its mean.

Internal semantic diversity. Within the assumed program syntax, every subsequence of program's instructions can be considered as a subprogram. Thus, by analogy to semantic diversity of entire programs, we can define *internal* semantic diversity, meant as the number of states programs can reach at a *certain point of execution* k (instruction number). This concept can be expressed in terms of traces as $|\{t^k : t \in \mathcal{T}\}|$, where $\mathcal{T}$ is a sample of traces. Figure 3 shows this quantity, as a function of instruction number k, for clarity only for program lengths $l = 10, 20$, and 30. The series start at abscissa $k = 1$, as $k = 0$ is here the state prior to program execution, which is always s_0. For technical reasons, we could here call SAMPLE with larger $n_p = 200,000,000$, which explains why the curves reach higher values.

Our instructions cannot reduce semantic diversity, i.e., produce fewer states than it is applied to. For each state in $M^{|M|}$, up to $|I| = 8$ states can be reached in a single step of program execution. Consequently, the curves in Fig. 3 have to be weakly monotonic. If we could consider *all* programs, this dependency would be strictly such. However, due to limited sample size, some divergence from monotonicity can be observed at the end of the curve for $l = 30$.

For the reasons explained earlier, the curve for $l = 20$ reaches the same level as that for $l = 30$: for the latter, it is much harder to generate programs that are nontrivial.

The flatness of the initial curve parts can be explained in the following way. Except for *neg*, the only instructions ca-

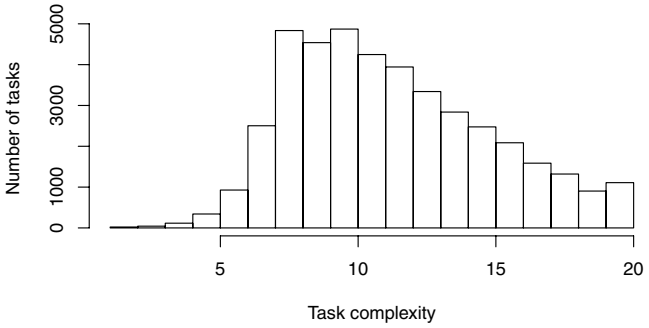

Figure 4: Distribution of task complexity.

pable of producing memory states that differ from the input state are the binary arithmetic operations, which pop two arguments from the stack and push only one result. With quite a shallow stack state at the beginning of program execution (two copies of the input integer), such instructions can quickly empty the stack and render the program trivial (see the footnote in Section 2.1). Therefore, to be nontrivial, a program has to constantly maintain sufficient stack depth, and the only instruction capable of this is *dup*. It can be then expected that many nontrivial programs have numerous *dup* instructions at the beginning. This implies fewer arithmetic operations at these locations, and causes the semantic diversity to grow relatively slowly.

Task complexity. Every program semantic (element of $\mathcal{S}$) generated by SAMPLE is a final state of at least one program trace (element of $\mathcal{T}$), and identifies a task. But it might be the case that the same state is being reached, and thus solved, by some other program(s) from the sample at an earlier stage of execution.

The length of the shortest program that generates data is a natural measure of complexity of that data, known as Kolmogorov complexity. An approximation of this complexity measure can be easily calculated from our random sample. For a given task $s \in \mathcal{S}$ (which is also a state), we find the index of the earliest position of this state in all traces, i.e., $\arg\min_k t^k : t^k = s, t \in \mathcal{T}$. This number characterizes the complexity of the task in the above sense. This is of course only an approximation (upper limit) of the actual Kolmogorov complexity, as an even shorter program that reaches s can still exists outside our sample.

Figure 4 presents the distribution of complexity of the sample of tasks $\mathcal{S}$ generated by SAMPLE for $l = 20$ and $n_t = 1,000,000$. Expectedly, most tasks in $\mathcal{S}$ turn out to be less complex than the assumed program length. Complexity of some of them is as low as 1, i.e., they can be solved by executing a single instruction. However, majority of tasks turn out to have complexity of 8 to 10. This makes us aware that complexity of tasks considered here can vary substantially, which will have certain implications for further analysis.

3. MODULARITY OF TASKS

In this section we attempt to identify certain properties of traces that could reveal and characterize modularity. In search for such properties, we recall the concept of functional module outlined in Introduction. In our formal framework, we identify functional module with a state in task trace that makes some parts of task independent or close-to-independent from others.

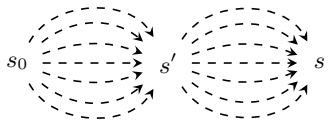

Figure 5: An exemplary trace of task s, for which s' is an intersection.

We formalize this as follows. An *intersection* for task s is any state $s' \neq s$ that belongs to $\bigcap_{p \in P(s)} t(p)$, i.e., is visited at least once by every program that solves s. Figure 5 presents an exemplary trace of task s, for which s' is an intersection. The edges are dashed to emphasize that the traces can visit other states before and after visiting s'. An intersection is then a common element for traces of all programs that solve a task.

If an intersection s' for task s exists, s becomes *decomposable*, and can be split into two *subtasks* (modules): (i) the subtask identified by s', and (ii) finding a program that implements a path from s' to s.[2] The intersection serves as a target for the former task, and as a starting state for latter task. Both subtasks can be solved independently, potentially at much lower computational cost (see Section 3.2).

However, it may be expected that intersections in strict sense do not exist for many tasks, especially when task trace contains many program traces. Nevertheless, for reasons that will be given later, any convergence of traces is desirable, even if not all of them meet at a single state. Therefore, we relax the formal definition, and aim at identifying states that form intersections in a weak sense, i.e., at least *some* of traces cross at them. Such states form 'waistlines' in task traces, like states s_4 and s_5 in Fig. 1. We propose two tools for characterization of such features in task traces.

3.1 Experiment 2: Diversity profiles

In this experiment, we search for signs of intersections by analyzing internal semantic diversity *within task trace*, in a way that resembles the experiment described in Section 2.1. By analogy to $t^i(p)$ that denotes the ith state in trace of program p, let $T^i(s)$ be the set of such ith states for all programs that solve s, i.e., $T^i(s) = \{t^i(p) : p \in P(s)\}$. $T^i(s)$ is then the set of states reached by all programs that solve s at the ith step of their execution. The cardinality of this set, $|T^i(s)|$, is the *semantic diversity of task trace* $T(s)$ at step i. This is analogous to semantic diversity as pictured in Fig. 3, however constrained to traces of a single task.

For instance, assuming that the programs p_1, p_2, p_3 considered in Fig. 1 are the only solutions to s, so that the figure depicts the entire $T(s)$, the diversity of $T(s)$ at step $i = 1, \ldots, 4$ amounts to $3, 2, 3$, and 1, respectively. The local minimum of diversity after the second instruction indicates the presence of intersection at state s_4.

This example shows that diversity of task trace can help discovering intersections for a *single* task. However, it cannot be directly applied to statistically characterize a *sample* of tasks. A task trace composed of m program traces can have diversity up to m. If the number of traces per task

[2]The latter subtask cannot be simply described as 'solving task s', as its initial state is s', not s_0.

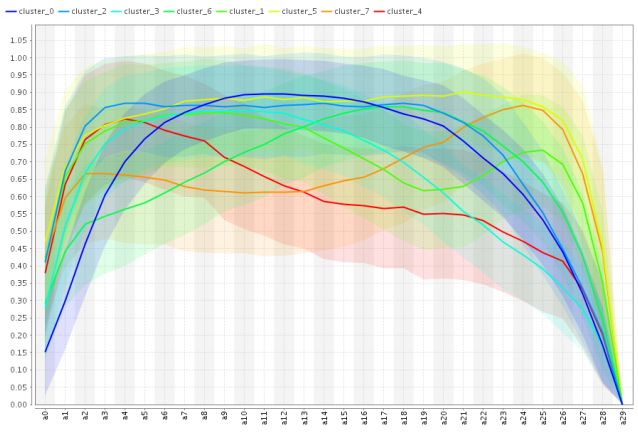

Figure 6: The prototypes of diversity profiles obtained using cluster analysis (abscissa: instruction number (step), ordinate: relative diversity). Shading reflects intra-cluster variance.

varies heavily in a sample (which is the case here), so will the diversity, and discovering any regularities will be hard.

Therefore, to make trace diversity comparable between tasks, we divide it by the maximum diversity of trace, defining so *relative diversity* at step i,

$$(|T^i(s)| - 1)/\max_{j=1,\ldots,l} |T^j(s)|$$

. It is a number in interval $[0, 1]$ that indicates how diverse is the task trace at specific instruction compared to the maximum diversity it reaches at any step. It drops to 0 only when all traces cross at a single state ($|T^i(s)| = 1$).

The material for analysis is here the sample generated by SAMPLE for $l = 30$ and $n_p = 200,000,000$. From that sample, we remove the tasks that have less than 3 program traces, because few traces cannot tell much about trace structure. Next, for each task, we calculate the relative diversity for consecutive steps $i = 1, \ldots, 30$, which results in a vector of 30 numbers. The overall outcome is a table of relative diversities that has 30 columns and as many rows as the number of considered tasks $|\mathcal{S}|$ (here: 6375). To identify the possible types of 'diversity profiles' of tasks, we cluster the rows of this table using the k-means algorithm, and Euclidean distance to measure similarity between rows. Figure 6 presents graphically the centers of clusters identified for $k = 8$, with the horizontal axis corresponding to instruction number. The translucent shading depicts variance. Table 1 gives the number of tasks assigned to each cluster.

The common feature of all profile patterns is that they tend to get wider for the few initial instructions (program prologue), and, conversely, narrower when approaching the end of a program (program epilogue). This was expected, as all traces of programs that solve the same task start at a single state (s_0), and have to meet at the same target state. Also, task diversity cannot vary arbitrarily from step to step, as the number of distinct states that can be reached from any state is small (cannot exceed 8, the number of instructions).

This is, however, where the similarities end. Cluster analysis clearly reveals a few distinct types of profiles. Between the prologue and epilog, they vary substantially: some of them maintain roughly the same diversity for the entire time

Table 1: Sizes of clusters shown in Fig. 6.

Cluster #	*Number of tasks*	*% of tasks*
0	931	15%
1	740	12%
2	928	15%
3	975	15%
4	540	8%
5	743	12%
6	939	15%
7	579	9%

of program execution, while other exhibit substantial narrowing at various stages of execution. The presence of the latter can indicate the existence of intersections. The figures in Table 1 show that such profiles are not a fluke, but embrace a significant fraction of tasks. The results for other program lengths, not shown here for brevity, revealed similar patterns. This suggests that traces of some tasks indeed tend to narrow at certain execution stages.

3.2 Experiment 3: Intersections

Although analysis of diversity profiles leads to interesting outcomes, it is naive in assuming that program traces intersect at the same stage of execution. Here, we abstract from absolute positions of instructions in the program sequence.

Let $t^-[s]$ denote the position of the earliest occurrence of state s in program trace t. Analogously, let $t^+[s]$ be the position of of the earliest occurrence of s *counted from the end* of trace t. For the trace of p_1 in Fig. 1, $t^-[s_1] = 1$ and $t^+[s_1] = 3$. As a trace can revisit the same state, $t^-[s] + t^+[s] \leq l$ (e.g., for trace of p_2 in Fig. 1, $t^-[s_1] = t^+[s_1] = 1$). If s does not occur in t, we assume $t^-[s] = t^+[s] = 0$.

Next, we define *centrality of state s' for task s* as:

$$c(s', s) = \frac{1}{|T(s)|(l/2)^2} \sum_{t \in T(s)} t^-[s']t^+[s'] \qquad (1)$$

The term in denominator serves normalization, so that $c(s', s) \in (0, 1]$. $c(s', s)$ attains the maximum value of 1 when two conditions are fulfilled: all traces in $T(s)$ traverse s, and, for each $t \in T(s)$, $t^-[s'] = t^+[s'] = \frac{l}{2}$, i.e., s 'halves' each trace. A state that is closer to s_0 or s, or such that not all traces pass through it, has lower centrality. For instance, for the example shown in Fig. 1 $c(s_4, s) = \frac{1}{3*2^2}(2{\cdot}2+2{\cdot}2+0{\cdot}0) = \frac{2}{3}$, while $c(s_1, s) = \frac{1}{3*2^2}(1{\cdot}3+0{\cdot}0+3{\cdot}1) = \frac{1}{2}$. $c(s_5, s)$ is even lower ($\frac{1}{3}$), as it occurs only in one trace. This clearly shows the advantage of centrality over the positional approach from Section 3.1: an intersection (s_1) can be valued more than a non-intersection (s_5) even if it does not group all traces in $T(s)$ at the same position.

Although both s_4 and s_1 occur in two traces, s_4 gets a higher value of $c()$ as its position in traces is more central. Centrality prefers intersections that are close-to-halfway in trace, because such intersections provide decompositions that lead to greatest computational gains. For instance, if all traces meet at an intersection after k instructions, the total cardinality of search spaces of subtasks is $|I|^k + |I|^{l-k}$, while for task it has $|I|^l$ programs. Decomposition leads to exponential reduction of search space cardinality, and such reduction is maximal for $k = \lfloor l/2 \rfloor$. In particular, a 'degenerate' decomposition (e.g., $k = 1$ or $k = l - 1$) will be strongly discouraged by $c()$.

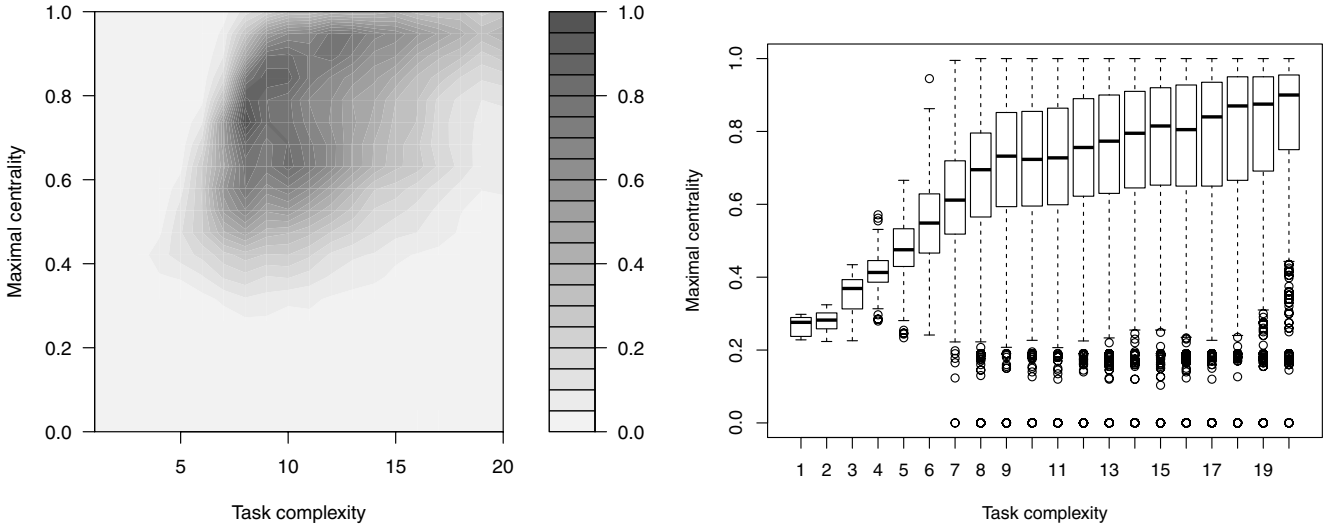

Figure 7: The distribution of tasks with respect to complexity (horizontal axis) and maximal centrality (vertical axis), shown as density graph (left) and box plot (right).

In this experiment, we consider the sample of traces analyzed in the previous section, i.e., with filtered out tasks that have two or less traces. For each task in the sample, we calculate the centrality for all states in its trace, and find its maximum. This value becomes a feature of task, intended to reflect its functional modularity (precisely, the most prominent and central narrowing in task trace).

In Fig. 7, we present the joint distribution of task complexity (as defined in Section 2.1) and maximal centrality. Both insets present the same data, the left one using a density plot, the right one using a box plot. These charts clearly reveal an almost monotonous tendency: maximal centrality tends to grow with task complexity. This is almost certainly not an artifact of our sampling method: program traces tend to intersect more frequently and centrally for complex tasks, despite the fact that for them we have fewer programs per task in the sample. This suggests that complex tasks may be more modular than the simpler ones, and is the main result of this study.

4. DISCUSSION

The spectrum of programming tasks revealed in the experiments can be characterized using concepts of modular interdependency [16]. A task that has an intersection in the strict sense is *separable*, and can be decomposed into two fully independent subtasks (Fig. 5). If more than one such intersection is present, this leads to more modules (e.g., three in Fig. 8a), each of them separable from all others. Assuming that the number of programs that solve the task remains constant, the greater number of separable modules, the easier it is to solve the entire task. This is easy to notice by considering an extreme degenerate case of a task with $l-1$ intersections. In such a case, traces of all programs that solve the task converge to single trace, which contains only intersections, and there are many instructions that move the interpreter from one intersection to the next. A mutation that replaces a single instruction with another that leads to the same state does not have to be compensated by another

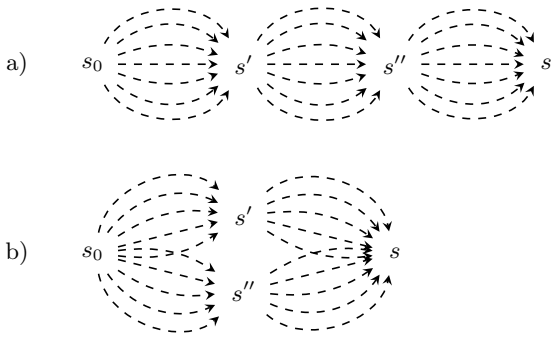

Figure 8: Other types of task traces with respect to modularity (cf. Fig. 1).

mutation in order to keep the program solving the task. There is no epistasis between the components of solutions.

If an intersection in the strict sense does not exist, but some traces do cross, a task can be considered *nearly-decomposable* [12], or *decomposable but not separable* [16] (p. 113). For such tasks, modification in one module may or may not require modification in another module in order to keep the program being a solution to the task. For instance, in Fig. 8b), modifying the left module (subprogram) can be neutral for its semantics (e.g., the trace of subprogram can still end with s'), in which case the right subprogram does not have to be changed. But if the semantic of left subprogram changes from s' to s'', the right subprogram is likely to have to change too, in order to keep the semantic of entire program unchanged (as it is unlikely that the unchanged right subprogram could lead from s'' to s). Such tasks are particularly interesting, as there is evidence that they are the frequent in natural systems [16], and can be hypothesized to constitute great share of real-world problems.

5. CONCLUSION

The general conclusion from this study is that there is substantial variation of semantic characteristic among program-

ming tasks. This variation concerns not only the distribution of program semantic and task complexity, but also how frequently the traces of programs that solve a task cross with each other and shape so modularity. Most importantly, the distribution of this characteristic, measured in terms of diversity profiles or centrality, is not uniform across the space of all tasks: there are evident patterns that point to existence of multiple types of tasks in terms of modularity and complexity.

Another conclusion is that, even in a random sample of tasks, many tasks can be modular. Whether separable or nearly-decomposable, they seem to constitute a substantial part of the universe of all tasks. Modularity could be then not only a feature of carefully designed artificial benchmarks (like NK-landscapes [3], HIFF [16], or [2]), or real-world problems, but an inherent property of many non-trivial programming tasks.

The approach we proposed and followed here is conceptually simple and minimalistic. It abstracts from any specific search algorithm and does not require a metric in the semantic space, as it is often practiced in studies on program semantics in GP (e.g., [8]). The only information exploited by this approach is indiscernibility of memory states. Thus, the conclusions drawn here can be considered problem-independent, or even domain-independent; at least, it is hard to see why they would not generalize to other loop-free stack-based languages and instruction sets.

On the other hand, our framework has certain limitations. To analyze the properties of a task, we require a sample of programs that solve it. However, when a search algorithm faces a specific task, it obviously has no solutions to it yet. If it works in an iterative mode, as it is the case for GP, it should gradually move toward them, but there is no guarantee it will ever find any. This raises a question to be addressed, i.e., whether analogous properties can be extracted from arbitrary samples of programs.

This is related to another observation, namely that the tools proposed here are applicable to any set of programs. In particular, an analogous analysis could be applied to a population of programs evolved by genetic programming. This should help answering questions like: Do the programs become more functionally modular with evolution time? In further perspective, this could also help designing methods (e.g., recombination operators) that exploit modularity for the sake of scalability and search effectiveness.

6. ACKNOWLEDGMENT

This work has been supported by grants no. DEC-2011/01/B/ST6/07318, 91-528/12-DS, and N N519 441939.

7. REFERENCES

[1] L. Altenberg. Modularity in evolution: Some low-level questions. In D. Rasskin-Gutman and W. Callebaut, editors, *Modularity: Understanding the Development and Evolution of Complex Natural Systems*, chapter 5, pages 99–128. MIT Press, Cambridge, MA, USA, June 2005.

[2] N. Kashtan and U. Alon. Spontaneous evolution of modularity and network motifs. *Proceedings of the National Academy of Sciences*, 102(39):13773–13778, Sept. 27 2005.

[3] S. A. Kauffman. *The Origins of Order: Self-Organization and Selection in Evolution*. Oxford University Press, USA, 1 edition, June 1993.

[4] K. Krawiec and B. Wieloch. Analysis of semantic modularity for genetic programming. *Foundations of Computing and Decision Sciences*, 34(4):265–285, 2009.

[5] W. Langdon and R. Poli. *Foundations of Genetic Programming*. Springer-Verlag, 2002.

[6] H. Majeed, C. Ryan, and R. M. A. Azad. Evaluating GP schema in context. In H.-G. Beyer et al., editors, *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, volume 2, pages 1773–1774, Washington DC, USA, 25-29 June 2005. ACM Press.

[7] N. F. McPhee, B. Ohs, and T. Hutchison. Semantic building blocks in genetic programming. In M. O'Neill, L. Vanneschi, S. Gustafson, A. I. E. Alcázar, I. D. Falco, A. D. Cioppa, and E. Tarantino, editors, *Genetic Programming*, volume 4971 of *LNCS*, pages 134–145. Springer, 2008.

[8] A. Moraglio. Geometry of evolutionary algorithms. In D. Whitley, editor, *GECCO 2011 Tutorials*, pages 1439–1468, Dublin, Ireland, 12-16 July 2011. ACM.

[9] A. Moraglio, K. Krawiec, and C. Johnson. Geometric semantic genetic programming. In C. Igel, P. K. Lehre, and C. Witt, editors, *The 5th workshop on Theory of Randomized Search Heuristics, ThRaSH'2011*, Copenhagen, Denmark, July 8-9 2011.

[10] M. O'Neill, A. Brabazon, and E. Hemberg. Tracer spectrum: a visualisation method for distributed evolutionary computation. *Genetic Programming and Evolvable Machines*, 12(2):161–171, June 2011.

[11] S. C. Roberts, D. Howard, and J. R. Koza. Evolving modules in genetic programming by subtree encapsulation. In J. F. Miller, M. Tomassini, P. L. Lanzi, C. Ryan, A. G. B. Tettamanzi, and W. B. Langdon, editors, *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 160–175, Lake Como, Italy, 18-20Apr. 2001. Springer-Verlag.

[12] H. Simon. *The Sciences of the Artificial*. MIT Press, Cambridge, MA, 1969.

[13] L. Spector, C. Perry, J. Klein, and M. Keijzer. Push 3.0 programming language description. Technical Report HC-CSTR-2004-02, School of Cognitive Science, Hampshire College, USA, 10 Sept. 2004.

[14] J. M. Swafford, E. Hemberg, M. O'Neill, M. Nicolau, and A. Brabazon. A non-destructive grammar modification approach to modularity in grammatical evolution. In N. Krasnogor et al., editors, *GECCO '11: Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 1411–1418, Dublin, Ireland, 12-16 July 2011. ACM.

[15] 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.

[16] R. A. Watson. *Compositional Evolution: The impact of Sex, Symbiosis and Modularity on the Gradualist Framework of Evolution*, volume NA of *Vienna series in theoretical biology*. MIT Press, February 2006.