Genet Program Evolvable Mach
DOI 10.1007/s10710-013-9200-2

Genetic programming: where meaning emerges from program code

Krzysztof Krawiec

Abstract Program behavior results from the interactions of instructions with data. In genetic programming, a substantial part of that behavior is not explicitly rewarded by fitness function, and thus emergent. This includes the intermediate memory states traversed by the executing programs. We argue that the potentially useful intermediate states can be detected and used to make evolutionary search more effective.

Keywords Genetic programming · Emergence · Semantics · Modularity · Interactions

In 'Genetic Programming and Emergence' [1] Wolfgang Banzhaf portrays a GP system as a hierarchy of components, where downward causation and upward causation lead to feedbacks that give rise to the overall dynamics of the system. That dynamics may involve various phenomena, among them such that have not been explicitly rewarded by selection pressure and thus can be deemed emergent, like modularity or bloat.

My argument starts with the observation that, if we agree with Banzhaf's perspective, which I find adequate, the *interactions* between the components should be considered more important than the components themselves. The components do not 'do' anything. Until one lets them interact, little can be said about their nature.

Supported by NCN grant DEC-2011/01/B/ST6/07318.

This comment refers to the article available at doi:10.1007/s10710-013-9196-7.

K. Krawiec (✉)
Institute of Computing Science, Poznan University of Technology, Poznan, Poland
e-mail: krawiec@cs.put.poznan.pl

Published online: 10 October 2013

Interactions take place at many levels within a GP system. Programs in a population interact with each other by competing for selection, and with the search operators by producing offspring. Instructions within programs interact by forming code fragments that contribute to program fitness. But there is one kind of interaction which I find particularly important: that between programs and data. Instructions interact with data by processing them and producing new data. From these interactions, the dynamics of a GP system originates, including the emergent phenomena. Together with the random variation, they are at the root of upward causation (see Table 1 in [1]).

The consequences of embracing this perspective are quite profound. Interactions are *functional* by nature: an interaction can be modeled as a function that maps the interacting components to an outcome of interaction. This is in stark contrast to the more *structural* perspective that prevails in GP, which focuses on the components: search operators, individuals, program syntax. The functional aspects, particularly those pertaining to the so important interactions between programs and data, attract less attention, sometimes because they are too difficult to analyze, or because they are regarded as too domain-specific. But a GP system cannot be fully understood without them, and I argue in the following that they are essential for making GP truly scalable.

The need for turning towards the functional perspective in bio-inspired computing has been articulated many times in the past (see, e.g., [5]). Within GP, the recent development of semantic genetic programming diverges from the structural perspective and adopts the functional one by focusing on the outcomes of interactions between program and data. Within that framework, program semantics is usually defined as the vector of outcomes of program execution for the particular training examples. The many-to-one mapping from the syntax (program code) to semantics (program meaning) allows expressing the same semantics in many ways, and thus involves the *causal slack* pondered over by Banzhaf [1]. Making search operators semantically-aware already proved a useful means for designing effective GP variants [6].

However, it would be incorrect to simply say that semantics *emerges* from program code as a whole. Programs in GP are being selected based on their fitness, which is directly derived from program semantics, and if we agree with Banzhaf, qualities for which entities are being *explicitly* selected should not be deemed emergent ([1], Section 4). Nevertheless—and this is the pivot of my argument—there is room for emergence *during* program execution. When run instruction by instruction, programs arrive at intermediate results, for which they are not explicitly rewarded. Consider the task of evolving a program that calculates the median of an array of numbers. A GP system solving that task can come up with programs that sort the array at a certain stage of execution. But nobody asked the programs to do that, nor selected them for this particular capability. This intermediate result emerged from the dynamics of the search process.

Why is this important? I claim that such functional intra-program phenomena can be harnessed. To start with, certain regularities emerge in intermediate products of program execution and can be exploited to improve search efficiency [3]. Also, the evolving programs can discover common intermediate semantic states and converge

to them [4], forming *functional modules*. In [2], we showed that some programming tasks are more modular in this sense than others. An effective exploitation of these phenomena remains to be seen, but seems attainable.

Is this a feature of GP only, or is it common to all automatic programming paradigms? I am inclined to adopt the former view, mostly due to the iterative nature of GP. Banzhaf rightly emphasized the temporal aspect: there are no feedbacks without time, and feedbacks are essential for emergence. This applies also to functional modules, which can emerge only when program induction is an iterative process. A GP system needs time to discover that certain intermediate outcomes are desirable. This, together with other features discussed in [1], makes GP quite a unique genre of automated programming, a genre that makes emergence not only possible, but employs it as one of its main vehicles.

References

1. W. Banzhaf, Genetic Programming and Emergence, Genetic Programming and Evolvable Machines (2013)
2. K. Krawiec, On relationships between semantic diversity, complexity and modularity of programming tasks. In Genetic and Evolutionary Computation Conference. New York, NY, USA. ACM, 783–790 (2012)
3. K. Krawiec, J. Swan, Pattern-Guided Genetic Programming. In *Genetic and Evolutionary Computation Conference*, New York, NY, USA. ACM (2013)
4. K. Krawiec, B. Wieloch, analysis of semantic modularity for genetic programming. Found. Comput. Decis. Sci. **34**(4), 265–285 (2009)
5. M. Mitchell, Ubiquity symposium: biological computation, *Ubiquity*, February, 17 (2011)
6. A. Moraglio, K. Krawiec, C.G. Johnson, Geometric semantic genetic programming. In *Parallel Problem Solving from Nature—PPSN XII*, eds. by Carlos A. Coello Coello et al. (Springer, Berlin, 2012), pp. 21–31