What, if anything, is a programming paradigm?

Robert Harper
Carnegie Mellon University

April, 2017

Everyone who writes about programming languages seeks to impose order on the chaos of extant languages. A common strategy is to borrow Thomas Kuhn's concept of a *scientific paradigm*, itself a not uncontroversial attempt to explain the social processes of science in his famous book entitled *The Structure of Scientific Revolutions* (Kuhn, 1975). Following Kuhn, observed software development practices are codified as *programming paradigms*, such as imperative, functional, object-oriented, concurrent, and logic programming, to name a few popular classifications. The trouble with programming paradigms is that it is rather difficult to say what one is, or how we know we have a new one. Without precise definitions, nothing precise can be said, and no conclusions can be drawn, either retrospectively or prospectively, about language design. Yet the convention of organizing languages into paradigms is so pervasive, I am asked to justify not adhering to it.

It is a matter of taxonomy versus genomics.

Stephen Jay Gould, in his remarkable essay entitled "What, if anything, is a zebra?" (Gould, 1983), criticizes cladistics, the classification of species by morphology. According to Gould, there are three species of black-and-white striped horse-like animals in the world, two of which are genetically closely related to each other, and one of which is not (any more so than to any other mammal). It seems that the mammalian genome encodes a propensity to display stripes that is expressed in disparate evolutionary contexts. From a genomic point of view, the clade of zebras can be said not to exist: there is no such thing as a zebra! It is more important to study the genome, and the evolutionary processes that influence it and are influenced by it, than it is to classify things based on morphology.

In writing *Practical Foundations for Programming Languages* I follow Gould in avoiding the distractions of morphology, focusing rather on the genes that express themselves in the structure of programs. The genomics of programming is type theory, the systematic study of computation. Originally formulated as a foundation for mathematics in which propositions are types and proofs of propositions are seen as programs of that type, the theory of types has become the central organizing principle of programming language research that unifies language semantics, program verification, and compiler implementation. Through deep connections to logic, algebra, and topology, type theory is the touchstone of truth in programming languages, isolating and clarifying the central concepts of computation. My account of the structure of programming languages ignores morphology, and focuses instead on type structure.

Paradigms are clades. Types are genes.

References

S.J. Gould. *Hen's Teeth and Horse's Toes*. Norton, 1983. ISBN 9780393311037. URL `https://books.google.ca/books?id=EPh9jD0XwR4C`.

Shriram Krishnamurthy. Teaching programming languages in a post-linnaean age. In *ACM SIGPLAN Workshop on Undergraduate Programming Language Curricula*, 2008. URL `https://cs.brown.edu/~sk/Publications/Papers/Published/sk-teach-pl-post-linnaean/`.

Thomas S Kuhn. *The Structure of Scientific Revolutions: 2d Ed., enl.* University of Chicago Press, 1975.