

# Categorical Semantics for Array-Based Programming Languages

Miles Gould  
University of Glasgow

December 13, 2007

## Background

Consider the usual addition operator between (real or complex) numbers:

$$a, b \mapsto a + b$$

This can be straightforwardly extended to give an additive structure on  $\mathbb{R}^n$  for any  $n \in \mathbb{N}$ :

$$(a_1, \dots, a_n) + (b_1, \dots, b_n) := (a_1 + b_1, \dots, a_n + b_n)$$

Similarly, we can extend our notion of addition to give a canonical way of adding arrays of any shape:

$$(a_{ijk\dots z}) + (b_{ijk\dots z}) := (a_{ijk\dots z} + b_{ijk\dots z})$$

More interestingly, we obtain a canonical way of adding a number to an array:

$$\begin{aligned} a + (b_{ijk\dots z}) &:= (a + b_{ijk\dots z}) \\ (a_{ijk\dots z}) + b &:= (a_{ijk\dots z} + b) \end{aligned}$$

or more generally, a way of adding two arrays of different shapes, providing that the shape of one is a prefix of the shape of the other:

$$(a_{ijk\dots m}) + (b_{ijk\dots z}) := (a_{ijk\dots m} + b_{ijk\dots z})$$

Such operations, which are defined simultaneously over arguments of many different types, are said to be *polymorphic*. Polymorphism is an important concept in computing, as it allows the same task to be done by a smaller and less repetitive (and thus more maintainable) body of code, and it allows existing code to be extended and applied in unforeseen ways. More importantly, it allows a higher-level and more general approach to problems, providing many of the same benefits in the field of computing as the axiomatic approach does in mathematics.

Polymorphism comes in many different flavours, which have been studied to varying degrees by mathematicians and computer scientists. The shape-based polymorphism described above has seen comparatively little study in an academic context, and yet it does form the basis of several important real-world programming languages.

The first of these is undoubtedly Iverson's APL (first described in [?], from which it takes its name), which is among the oldest computer programming languages in continued use. Like MacCarthy's Lisp (which is of similar vintage), it was originally intended as an alternative notation for mathematics, but turned out to be more useful as a tool for describing algorithms to computers. APL's descendents include J, K, A+, Matlab and IDL: they are extensively used in the worlds of engineering, scientific modelling, and finance (the myth that APL stood for "actuarial programming language" was once common, and A+ was developed in-house at Morgan Stanley). They have several advantages over conventional von Neumann programming languages, among them:

1. Concision: even without use of Iverson's eye-wateringly terse notation, array-based programs are often many times shorter than equivalent programs written in conventional languages. The array-based paradigm often allows for radically different and dramatically simpler approaches to solving problems. My (informal and small-scale) experiments suggest that they are also usually slightly shorter than equivalent programs written in functional languages. This concision (their advocates claim) allows fewer programmers to solve harder problems in less time, and with fewer maintenance headaches.
2. Array-based programs are usually so highly data-parallel that they can be trivially ported to take advantage of multi-processor or multi-core machines.
3. APL and its descendents have a proven track record of being useful to non-programmers.

## Research Programme

1. Learn more about APL and its relatives.
2. Learn more about semantics for programming languages.
3. Go to some APL conferences.
4. Develop a categorical semantics for a simple APL-like language, with a view towards full abstraction or some other buzzwordy desideratum.
5. Get academics to take the APL family seriously.
6. Get APLers to take category theory seriously.

7. Contribute to a greater understanding and less duplication of work between the functional-programming crowd and the array-programming crowd.
8. ???
9. Profit!