

Categorical Semantics for Array Programming Languages

Miles Gould
University of Glasgow

December 17, 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.

Such languages are usually called “array programming languages”. The first of these is undoubtedly Iverson’s APL (first described in [2], from which it takes its name), which is among the oldest computer programming languages in continued use. Like McCarthy’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).

The APL family implement the shape-based polymorphism described above in a very straightforward way, as the following example session with a J interpreter shows. Lines starting with “NB.” are comments.

```

NB. Scalar addition
1 + 3
4
NB. Vector addition
1 2 3 + 4 5 6
5 7 9
NB. Adding vectors to scalars
1 + 4 5 6
5 6 7
NB. Addition of arrays: i.3 is the array 0 1 2
NB. and (3 3 $ 0) is a 3x3 array of zeroes.
(i.3) + (3 3 $ 0)
0 0 0
1 1 1
2 2 2
NB. Addition along other axes can be achieved
NB. using the " (rank) conjunction
(i.3) (+"1) 3 3 $ 0
0 1 2
0 1 2
0 1 2
NB. The same rules apply for user-defined functions.
NB. Define avg to return the mean of the entries of its argument.
avg =. +/ % #
avg 1 2 3 4 5
3
3 4 $ i.12

```

```

0 1 2 3
4 5 6 7
8 9 10 11
    avg 3 4 $ i.12
4 5 6 7
    (avg"1) 3 4 $ i.12
1.5 5.5 9.5

```

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 and shape polymorphism often allow for radically different and dramatically simpler approaches to solving problems. My (informal and small-scale) experiments suggest that array-based programs 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. Generality: the array-based approach allows for the use of high-level tools not available in other languages, such as a general outer product operator.
3. Parallelism: Array-based programs are typically highly data-parallel, and can be trivially ported to take advantage of multi-processor or multi-core machines.
4. APL and its descendents have a proven track record of being useful to non-programmers.

Research Programme

The aim of this project would be to describe the shape-based polymorphism exhibited by the APL family of languages in a categorical way, and thence

- to construct a formal semantics for a simple APL-like language;
- to explore the connections between APL-style polymorphism and more conventional kinds;
- to apply the current body of knowledge on programming language semantics in this new setting, and prove new and hopefully useful results;
- to expose shape-based polymorphism to a wider academic audience;
- to expose the fruits of modern computer science to the APL community.

Category theory is the preferred way of approaching the semantics of programming languages and their type systems, and thus the obvious approach to use. Dalhousie has a world-class category theory group, and Peter Selinger is an expert in its applications to computer science.

I have been able to find very little previous research on this problem; Jay has done categorical work on a form of shape-based polymorphism (see [3] for an overview of his approach), but it is not yet clear to me how (or if) his work relates to the polymorphism exhibited by APL code. In particular, his work concentrates on programs for which the shape of the output can be determined at compile-time, which is not the case for many interesting APL programs. It is also unclear how such things as the rank conjunction and the shape function (both exhibited above) might fit into his framework. Attempts to construct a mathematical framework for APL within the community (such as [1]) have so far had little success.

Significance of this Research

The APL community is in an unusual position: largely forgotten and outside the mainstream, they still have much to teach the larger computer science community. Indeed, the noted computer scientist Guy Steele gave a keynote speech at SIGAPL 2007 entitled “What APL can teach the world”. Conversely, they are in a position to benefit from recent developments in other areas, which their isolation has kept from them. For instance, many of the problems that have dogged attempts to produce APL compilers (such as the presence of dynamic typing, reflection, and evaluation of code generated at runtime) have long been solved in the Lisp world. It is my hope that a modern mathematical treatment of APL will bring its unusual nature and good features to wider attention, and help to enable dialogue between the two communities.

My research thus far has been in the intersection of higher category theory and universal algebra; which is to say, I have been studying general ways in which simple operations can be composed into more complicated ones. I believe that this makes me ideally prepared for this research programme.

References

- [1] Phil Chastney. Formal semantics of APL: a review of initial findings. *SIGAPL APL Quote Quad*, 32(4):74–82, 2002.
- [2] Kenneth E. Iverson. *A programming language*. John Wiley & Sons, Inc., New York, NY, USA, 1962.
- [3] C.B. Jay. Shape in computing. *ACM Computing Surveys*, 28(2):355–357, 1996.