UNIVERSITY OF CAMBRIDGE

Escaping the Pointless
Embedding Pointful Array Programming in Python

Jakub Bachurski

Project Supervisor: Alan Mycroft

Python & NumPy

Python is a **dynamic**, **interpreted** and **imperative** language.

NumPy is a **high-performance** library for **array programming** in the model of APL [Iverson, 1962].

```
c = numpy.empty((len(a[0]), len(a)))
for i in range(len(a[0])):
    for j in range(len(a)):
        c[i, j] = a[j, i] * b[j]
```

```
c = numpy.add(
    numpy.transpose(a, (1, 0)),
    numpy.expand_dims(b, 1)
)
```

NumPy is highly influential, especially in deep learning – **TensorFlow**, **PyTorch**, **Jax**.

Problems and alternatives

NumPy and similar approaches yield **fast** programs relatively easily, but also suffer from problems.

- **Difficult to work with** due to excessive flexibility
- **Many primitives** and shape-polymorphism, making static analysis difficult
- **Unintuitive** notation for higher-dimensional computations

Can we give a domain-specific language that improves on these and is just as efficient? In this work, we consider an index-oriented **(pointful)** approach, which allows writing array comprehensions:

```
c = array(lambda i, j: a[j, i] * b[j])
```

This is in contrast to NumPy, where no efficient loop-like constructs exist (the **point-free** style), and **whole-array** operations are the focus.

We introduce a shallow-embedded Python DSL – *Ein* – and base it on a calculus we call *Phi*.

Array programming in NumPy

1. Manipulate array shape to prepare operation – `transpose, unsqueeze, reshape`
2. Use a flexible API primitive – `add, mean, matmul, meshgrid` (hundreds of these!)

We only consider **rectangular** and **homogenous** n-dimensional arrays of primitive data types. We say that such an array has n **axes** indexed from 0, each having a **size**.

$$\operatorname{shape}\left(\begin{bmatrix}1&2&3\\4&5&6\end{bmatrix}\right) = (\underset{\text{axis 0}}{2}, \underset{\text{axis 1}}{3})$$

Broadcasting

Elementwise operations concern arrays of the same **shape**:

$$\begin{bmatrix}1&2\\3&4\end{bmatrix} + \begin{bmatrix}1&-1\\-1&1\end{bmatrix} = \begin{bmatrix}1+1&2-1\\3-1&4+1\end{bmatrix} = \begin{bmatrix}2&1\\2&5\end{bmatrix}$$

NumPy efficiently generalises them via to arrays with corresponding axes of size 1. We repeat values along such axes to give compatible sizes.

$$\underbrace{\begin{bmatrix}1&2&3\\4&5&6\end{bmatrix}}_{\text{shape }(2,3)} \cdot \underbrace{\begin{bmatrix}1\\-1\end{bmatrix}}_{(2,1)} = \begin{bmatrix}1&2&3\\4&5&6\end{bmatrix} \cdot \begin{bmatrix}1&1&1\\-1&-1&-1\end{bmatrix} = \begin{bmatrix}1&2&3\\-4&-5&-6\end{bmatrix}$$

$$\underbrace{\begin{bmatrix}1&2&3\\4&5&6\end{bmatrix}}_{(2,3)} + \underbrace{\begin{bmatrix}1&0&-1\end{bmatrix}}_{(1,3)} = \begin{bmatrix}1&2&3\\4&5&6\end{bmatrix} + \begin{bmatrix}1&0&-1\\1&0&-1\end{bmatrix} = \begin{bmatrix}2&2&2\\5&5&5\end{bmatrix}$$

Shape manipulation

To perform the right operation, we usually first manipulate the shape of operand arrays.

```
numpy.zeros((7, 11, 13)).shape    # -> (7, 11, 13)
numpy.transpose(numpy.zeros((7, 11, 13)), (2, 0, 1)).shape  # -> (13, 7, 11)
numpy.arange(5)  # -> [0, 1, 2, 3, 4]
numpy.expand_dims(numpy.arange(5), (0, 2)).shape  # -> (1, 5, 1)
```

Pointful Array Programming

We introduce the Phi calculus, based on $\tilde{F}$ [Shaikhha et al., 2019].

$$\begin{aligned}
e ::= \quad & \operatorname{array} e\,(i \Rightarrow e) && \text{(array comprehension)}\\
\mid \quad & e[e] \mid \operatorname{size}_c(e) && \text{(indexing, array size along axis } c \in \mathbb{N})\\
\mid \quad & \operatorname{fold} e\,e\,(\langle x, x\rangle \Rightarrow e) && \text{(indexed fold)}\\
\mid \quad & \operatorname{let} x = e \operatorname{in} e && \text{(non-recursive let binding)}\\
\mid \quad & x \mid i \mid v && \text{(variable, index, constant)}\\
\mid \quad & \pi(e, \ldots, e) && \text{(scalar operator)}
\end{aligned}$$

$$\tau ::= \operatorname{Scalar} \mid \operatorname{Array} \tau \qquad \operatorname{Scalar} = \mathbb{B} \mid \mathbb{Z} \mid \mathbb{R}$$

For example: $\operatorname{matmul}(a, b) = \operatorname{let} n = \operatorname{size}_0(a) \operatorname{in} \operatorname{let} m = \operatorname{size}_1(a) \operatorname{in} \operatorname{let} k = \operatorname{size}_1(b) \operatorname{in}$
$\operatorname{array} n\,(i \Rightarrow \operatorname{array} k\,(j \Rightarrow \operatorname{fold} m\,0.0\,(\langle t, x\rangle \Rightarrow x + a[i][t] \cdot b[t][j])))$

Typing

Variables x and indices i belong in **separate environments** Γ, Δ.

$$\frac{\Gamma; \diamond \vdash n : \mathbb{Z} \qquad \Gamma; \Delta, i \vdash e : \tau}{\Gamma; \Delta \vdash \operatorname{array} n\,(i \Rightarrow e) : \operatorname{Array} \tau} \qquad \frac{\Gamma; \diamond \vdash n : \mathbb{Z} \qquad \Gamma; \Delta \vdash a : \tau \qquad \Gamma, k : \mathbb{Z}, x : \tau; \Delta \vdash e : \tau}{\Gamma; \Delta \vdash \operatorname{fold} n\,a\,(\langle k, x\rangle \Rightarrow e) : \tau}$$

Constraining sizes with $\Delta = \diamond$ **ensures** arrays remain **rectangular** and data parallelism is **regular**.

Bridging the pointful & pointless

Consider denotations of expressions in Phi (which is total and pure):

$$[\![-]\!] : \operatorname{Env} \to \operatorname{Value}$$

Since $\operatorname{Env} \simeq \operatorname{VarEnv} \times \operatorname{IndexEnv}$:

$$[\![-]\!] : \operatorname{Env} \to \operatorname{Value} \simeq \operatorname{VarEnv} \times \operatorname{IndexEnv} \to \operatorname{Value} \simeq \operatorname{VarEnv} \to (\operatorname{IndexEnv} \to \operatorname{Value})$$

Since indices span ranges $[0, n)$, there is a correspondence with n-dimensional arrays:

$$\operatorname{IndexEnv} \to \operatorname{Value} \;\simeq\; \operatorname{NDArray} \operatorname{Value}$$

This exposes the fact we need not *loop* over all assignments IndexEnv and can instead operate on entire arrays at a time. This corresponds to **broadcasting**!

Compiling by lifting into the Axial applicative

We introduce the **Axial applicative** which formalises a **direct connection** between **pointful** and **point-free** array programming.

Applicatives [McBride and Paterson, 2008] are a prevalent functional programming pattern which generalises monads. **Axial** is a **generalisation** of the **List and ZipList applicatives**.

An $\operatorname{Axial} \alpha$ is a function from **assignments** to a given set of **sized axes** into α. We show it is an applicative functor by defining pure and lift.

$$\operatorname{pure} x = \operatorname{Axial}(\varnothing, \lambda\, \iota.\, x)$$

The application lift f corresponds to a **broadcast** binary operation f:

$$\operatorname{lift} f\, a\, b = \operatorname{Axial}(\operatorname{axes}(a) \cup \operatorname{axes}(b), \lambda\, \iota.\, f(a(\iota), b(\iota)))$$

An axial can be represented as an array, where each dimension corresponds to an axis.

Our novel array compiler $\mathcal{A}$ works recursively by keeping track of axial representations:

$$\mathcal{A}\left(\begin{array}{ccccccc} & & & + & & & \\ & & \times & & & ! & \\ & ! & & ! & & ! & \mathbf{i} \\ a & \mathbf{i} & b & \mathbf{j} & c & \mathbf{j} & \end{array}\right) = \texttt{add}\left(\underbrace{\mathcal{A}\left(\begin{array}{ccccc} & & \times & & \\ & ! & & ! & \\ a & \mathbf{i} & b & \mathbf{j} & \end{array}\right)}_{\text{axes}\to(\mathbf{i},\mathbf{j})}, \texttt{transpose}\left(\underbrace{\mathcal{A}\left(\begin{array}{ccc} ! & & \\ ! & & \mathbf{i} \\ c & & \mathbf{j} \end{array}\right)}_{\text{axes}\to(\mathbf{j},\mathbf{i})}, (1,0)\right)\right)$$

Ein in practice

Excerpt from a graph neural network architecture (GAT) in NumPy:

```
logits = (
    numpy.transpose(s[..., numpy.newaxis], (0, 2, 1, 3))  # + [B, H, N, 1]
    + numpy.transpose(t[..., numpy.newaxis], (0, 2, 3, 1))  # + [B, H, 1, N]
    + numpy.transpose(e, (0, 3, 1, 2))  # + [B, H, N, N]
    + g[..., numpy.newaxis, numpy.newaxis]  # + [B, H, 1, 1]
)  # = [B, H, N, N]
```

With Ein, this code can be written more concisely:

```
logits = array(
    lambda b, h, u, v: s[b, u, h] + t[b, v, h] + e[b, u, v, h] + g[b, h]
)
```

Selecting NumPy primitives in Ein

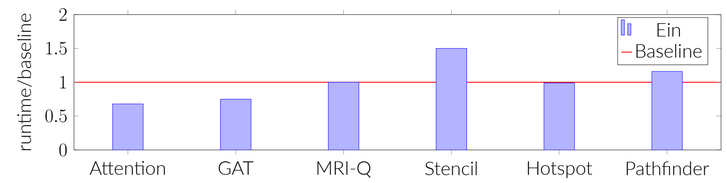
When **Ein** compiles programs into **NumPy**, its **primitives** (both general and specialised) are treated as machine instructions. Implementing Ein involves a variant of **instruction selection**.

```
# Indexing
array(lambda k, i: t[k, js[i]])          numpy.take(t, js, axis=1)
array(lambda i: a[max(i - 1, 0)])        numpy.pad(a[1:], ((1, 0)), mode='edge')
array(lambda i: m[i, i])                 numpy.diag(m)

# Tensor products
array(lambda p,i,j: sum(lambda x,y: (
    a[p, i, x, y] * b[y, x, j]           numpy.tensordot(a, b, ((2, 3), (1, 0)))
))
```

Benchmarks

Benchmarking the current **Ein** implementation against baseline **NumPy code** yields:



Examples sourced from: deep-learning architecture implementations, Parboil, Rodinia.

Previous work

- A practical pointful array programming language is **Dex**, but it uses a bespoke compiler.
- There are **limited deep-embedded** DSLs with pointful features, such as `einops`.
- An alternative shallow-embedded approach is **naming axes**, but the benefits are limited.
- We establish a **new formal connection** between pointful and point-free array programming. In Ein, loop-based **array comprehensions** are **compiled away** to **whole-array operations**.

Summary

- Based on previous work, the **Phi calculus** is introduced.
- Ein is a **DS L shallow-embedded in Python** which uses this calculus.
- By lifting computations into the **Axial applicative**, we can use **array programming** techniques.
- Ein utilises this approach, setting **NumPy as a compilation target**.
- The compiler can be **retargeted** to other frameworks **(Jax, PyTorch)** that dominate the deep-learning research landscape, integrating it with real-world applications.

kbachurski@gmail.com | jkb55@cam.ac.uk