Points for Free

Embedding Pointful Array Programming in Python

Jakub Bachurski, Trinity College

Computer Science Tripos, Part II

2023–2024

A dissertation submitted to the University of Cambridge
in partial fulfilment of the requirements for the degree of
Bachelor of Arts in Computer Science

Declaration of originality

I, Jakub Bachurski of Trinity College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose. In preparation of this dissertation I did not use text from AI-assisted platforms generating natural language answers to user queries, including but not limited to ChatGPT. I am content for my dissertation to be made available to the students and staff of the University.

Signed *Jakub Bachurski*
Date *September 29, 2024*

Proforma

Candidate Number	2405E
Title	Embedding Pointful Array Programming in Python
Examination	Computer Science Tripos – Part II, 2024
Word Count	11992[*]
Code Line Count	7071[†]
Project Originator	The candidate
Project Supervisor	Professor Alan Mycroft

Original aims of the project

Machine learning and scientific computing ecosystems are dominated by implementations of the *array programming model* in Python, such as NumPy. Writing and maintaining programs in the model is known to be difficult. However, significant engineering effort has been spent on making it efficient.

This project set out to investigate *pointful array programming* as an alternative by designing and implementing a pointful array language embedded in Python. Furthermore, the project would explore methods to execute it with Python's established array libraries. Thus, it would reconcile the expressiveness of pointful array programming, and performance of the array programming model.

Work completed

The project was a complete success. All success criteria were met with greatly extended scope, with the designed embedded language – Ein – implementing various array programming features previously unavailable in Python. An efficient compilation scheme from pointful array programming to the array programming model was developed, relying on a new formal connection between the two styles.

My project won first prize among undergraduates in the ACM Student Research Competition at the 51st POPL conference. Furthermore, a research paper coauthored with my supervisor was accepted for publication in the 10th ACM ARRAY proceedings.

Special difficulties

None.

[*] Computed with `texcount`.
[†] Computed with `cloc`.

Contents

1 Introduction

In his Turing Award Lecture, *Notation as a Tool of Thought*, Iverson argues for the importance of programming language design through the lens of mathematical notation [21]. The *array programming model* he introduced in APL is still in use today thanks to its performance and conciseness. The ideas presented in APL were revolutionary – particularly the brevity of the language and expressiveness, despite its relatively simple grammar. It contrasts in many ways with numerical languages of its time, such as FORTRAN – as Smillie describes in *Discovering Array Languages* [34].

APL's ideas certainly stood the test of time in the languages it influenced, such as J or MatLab. But this begs the question: is the model still appropriate for use in modern applications – such as deep learning – or is there need for a new notation?

1.1 Motivation

When I found myself in a team working on generalisation of deep learning models, I was shocked to see novel model architectures attract Python implementations as troublesome as this one, paraphrased below in the style of *NumPy:*

```
u = expand_dims(u, axis=-1)
v = expand_dims(v, axis=-1)
g = expand_dims(g, axis=-1)
logits = (
    transpose(u, (0, 2, 1, 3)) +  # + [B, H, N, 1]
    transpose(v, (0, 2, 3, 1)) +  # + [B, H, 1, N]
    transpose(e, (0, 3, 1, 2)) +  # + [B, H, N, N]
    expand_dims(g, axis=-1)       # + [B, H, 1, 1]
)                                 # = [B, H, N, N]
```

Problems found in this snippet are not uncommon in the domain. The code was difficult to write and is not much easier to read, there are plenty of cryptic parameters that are *probably correct*, and it was necessary to add comments. These issues can be attributed to a failure of the underlying paradigm – the array programming model. Paszke et al. [29] argue why the model might not be an apt abstraction for modern workflows. Though there are certainly benefits, such as the abundant parallelism present and usually good behaviour under automatic differentiation, there are also shortcomings. The notation is often too explicit while also being too unconstrained, leading to code unreadable by both humans and machines.

Worse yet, the deep learning ecosystem is extremely centralised – most research and engineering takes place in Python in a select few libraries. Examples include PyTorch, TensorFlow, JAX, and they all turn out to derive from the aforementioned NumPy [1, 11, 28], which itself embraces the array programming model [16]. The deep learning ecosystem is constantly evolving, with new operations and hardware targets constantly developed. This makes for an immense engineering effort, slowing down development of new practical approaches.

1.2 Prior art

Typical examples of languages used for numeric programming include C/C++ and FORTRAN. They tend to have influences from APL (in their design [6] or packages [14]), but are primarily imperative languages – which grant a higher degree of control at the expense of more complex compilers. Such languages are seldom applied directly in domains like deep learning by practitioners (unless necessary), and rather they are called from Python via its foreign function interface.

Issues of the array programming model have not gone unnoticed in the Python community. Similar to Iverson's approach, mathematical notation was sought after as an inspiration for an alternative – in this case this inspiration was *Einstein summation*. Initially limited approaches in Python – such as *Tensor Comprehensions* [35] and `einops` [31] – can be seen as the emergence of **pointful array programming**, as described by Paszke et al. [29] in the Dex language.

1.3 Aims

The main aim of the project is to design and implement a domain-specific language, hereafter called **Ein**, which pragmatically addresses problems of established Python array libraries by incorporating pointful array programming. This means:

- Ein needs to show improvement of notation on a selection of problems. It should be general enough to avoid awkward code switching and build on a formal foundation. There is a tool for every problem, and we should improve on the array programming model where it falls short.
- Since Python is such an important aspect of array programming in practice, Ein needs to be easily integrated with it.
- Ein should capitalise on existing engineering efforts, which already produces efficient array code targeting a variety of hardware.

I thus chose to embed Ein in Python, and execute it by calling NumPy routines.

Foreshadowing The motivating example translates to the following code using Ein:

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

The code became much closer to an index-oriented mathematical notation. It executes just as fast as the original by calling the same routines as the original, while being much more readable.

2 Preparation

This project inherently covers many different areas, as it aims to use methods of programming languages to solve problems of array programming practitioners (e.g. in machine learning). In this chapter:

- We begin with a description of the array programming model in the context of Python (Section 2.1).
- We then consider the pointful array paradigm and its inspirations in Section 2.2.
- Section 2.3 looks into the topic of embedding domain-specific languages.
- We introduce classic functional programming (Section 2.4) and compiler techniques (Section 2.5).
- Lastly, I give a project requirements analysis (Section 2.6) and starting point (Section 2.7).

2.1 Array programming model

Much of today's deep learning and scientific computing workflows takes place in the *array programming model* of Iverson [20]. This style is dominated by **whole-array operations**. The leading Python library for efficiently processing multidimensional arrays, NumPy, is no exception [16]. NumPy focuses on CPU execution and is implemented in highly-optimised C, playing a central role in numeric programming across the entire Python ecosystem. NumPy's design is motivated by Python's significant runtime overheads – it is profitable to offload large units of work to C. The core data structure is the `ndarray` – a multidimensional rectangular array of primitive values (usually fixed-sized integers and floats). We simply call these *arrays*.

Many modern Python array libraries are based on NumPy, including ubiquitous deep learning frameworks like PyTorch, TensorFlow, and JAX [1, 11, 28], which take advantage of hardware accelerators like GPUs. This similarity is seen as a merit, though many inconsistencies hinder interoperability [27].

We now introduce common terms for dealing with arrays. The number of dimensions of an array is called its *rank*. We call arrays of rank 0 – scalars, rank 1 – vectors, and rank 2 – matrices. Rectangular arrays have a consistent size in every dimension (*axis*), and as such have a *shape*, which is a tuple of natural numbers the same length as the rank. Indexing into an array a of shape $(d_0, ...d_{k-1}) \in \mathbb{N}^k$ is defined for indices $(i_0, ..., i_{k-1}) \in \mathbb{N}^k$ such that $0 \leq i_p < d_p$, and is written $a[i_0, ..., i_{k-1}]$. Axes are indexed from 0, and here we say that i_p indexes into axis p.

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \text{ is a rectangular array, but } B = \begin{bmatrix} 0 & 1 & \\ 0 & 1 & 2 \end{bmatrix} \text{ is not.}$$

$$\text{shape}(A) = (2, 3) \quad \text{rank}(A) = 2 \quad A[0, 2] = 3$$

Figure 2.1: Examples of array concepts

2.1.1 Programming in NumPy

We now give a summary of the key features of NumPy. Efficiency of many of the following primitives relies on the use of **strides** in the `ndarray` representation [16] – we treat this as an implementation detail. For clarity throughout this section, functions corresponding to NumPy primitives are written in `monospace`.

Broadcasting The obvious kind of whole-array operation is an **elementwise operator**:

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

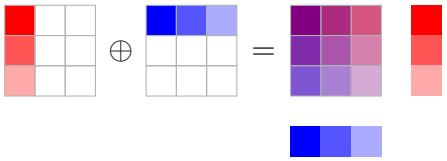
Pointfully, we define the action $C = A + B$ on matrices as $C[i,j] = A[i,j] + B[i,j]$ for all valid i, j.

But what about the cases where arrays do not have matching ranks? Consider scaling a matrix, i.e. $L = \lambda A$, defined $L[i,j] = \lambda \cdot A[i,j]$. **Broadcasting** generalises elementwise operations to the case where only a subset of axes is present. NumPy approaches this by *repeating axes of size 1* (1-axes), unambiguously indexed with 0. Where arrays have mismatched ranks, the shapes are prefixed with 1-axes. For instance, consider the outer product $C = a \otimes b$ ($C[i,j] = a[i] \cdot b[j]$). To compute it in NumPy, we shape a and b as a row and column vector respectively, so that $\text{shape}(a) = (n, 1)$ and $\text{shape}(b) = (1, m)$. Then:

$$C = \texttt{multiply}(a, b) \iff C[i,j] = a[i,j] \cdot b[i,j] = a[i,0] \cdot b[0,j]$$

This can be concise and elegant, but the main drawback is that broadcasting is somewhat ad-hoc and thus difficult to formalise. The condition for matching up axes of size d and d' is $d = d' \vee d = 1 \vee d' = 1$ – and such disjunctions are widely known to be difficult to handle in e.g. type inference.

A broadcasting (of a 'colour mixing' operator $\oplus$) in the style of an outer product is illustrated below:



Shape manipulation But how do we manipulate arrays into a form suitable for broadcasting? NumPy offers various primitives that change the shape of an array without copying its data (thanks to using strides). Say that a and b were vectors of shapes (n) and (m). Then we may use `expand_dims` to add the 1-axes:

$$\text{shape}(a) = (n) \implies \text{shape}(\texttt{expand\_dims}(a, \texttt{axis=1})) = (n, 1)$$
$$C = a \otimes b = \texttt{multiply}(\texttt{expand\_dims}(a, \texttt{axis=1}), \texttt{expand\_dims}(b, \texttt{axis=0}))$$

It is worth noting that the inverse of `expand_dims` (also called `unsqueeze`) is `squeeze`.

Now consider $C = A + A^T$ ($C[i,j] = A[i,j] + A[j,i]$). To permute axes of A, we use `transpose`:

$$A^T = \texttt{transpose}(A, (1,0)) \iff A^T[i,j] = A[j,i]$$
$$C = A + A^T = \texttt{add}(A, A^T) = \texttt{add}(A, \texttt{transpose}(A, (1,0)))$$

All of these primitives generalise to multiple dimensions. The main source of problems are the axis indices and permutations, which get harder to reason about as we generalise to more and more dimensions.

$S =$  $S[0,2] =$  $\texttt{transpose}(S, (1,0)) = S^T =$  $S^T[0,2] = S[2,0] =$ 

Reductions Operations we have considered so far cannot *accumulate* data. Though the paradigm does not forbid simply looping in Python, the idiomatic (and faster) approach is a *reduction*. To compute a so-called tropical matrix product,[1] we use `numpy.min`, parametrised by the index of the axis to reduce over:

$$C[i,j] = \min_k A[i,k] + B[k,j]$$

$$C = \texttt{min}(\texttt{add}(\texttt{expand\_dims}(A, \texttt{axis=1}), \texttt{expand\_dims}(A, \texttt{axis=0})), \texttt{axis=1})$$

Generality NumPy could be called a **first-order** interface, since its primitives cannot be parametrised with functions. Thus, we can only broadcast and reduce with some specialised operations, leading to performance limitations. If not for `numpy.argmax`, one would be forced to use a slow Python loop:

```
p = 0
for i in range(len(a)):
    if a[i] > a[p]: p = i
```

Though there exist more efficient implementations of the above routine, they are necessarily slower than a native (e.g. C++) implementation due to Python's dynamic typing.

2.1.2 Jagged arrays

Jagged (non-rectangular) arrays are used less often than their counterpart. They cause irregular parallelism, which is harder to implement efficiently. NumPy-like libraries forbid them entirely in their `ndarray` type. This is not unprecedented – the same constraint is present in the Futhark array language [18], and preservation of rectangular arrays can be seen as one of the core features of the dependent type system in Dex. We shall also consider jagged arrays to be an error.

2.1.3 Types

Python supports a form of optional typing through *type hints* (*annotations*), written `name: type`. The signature `(x: int, t: str) -> str` would then indicate a function which takes an integer `x` and a string `t`. One can (and should) take advantage of this system to document the behaviour of functions and allow some static analysis. Type checking is facilitated via third party tools, with `mypy` being the most popular.

Unfortunately, static typing is notoriously difficult in the array programming model. In general, keeping track of array sizes already invites dependent types [17]. Consider typing for array element types and ranks. There are many problems that such a type system faces in NumPy, but primarily:

- An extremely broad API, with arrays often constructed arbitrarily with no underlying principles.
- Complex dependencies between types. For instance, broadcasting of arrays of ranks k and ℓ produces a rank $\max(k, \ell)$. Similarly, un/squeezing 1-dimensions increments/decrements the rank. Type-level arithmetic like this is generally difficult to encode.
- Even element types are difficult to model, as they are often dependent on arguments (e.g. `dtype`) to NumPy routines. Otherwise, they follow C-like type promotion, which also complicates inference.

[1] The tropical $(\min, +)$ algebra is useful in various shortest path problems on graphs.

```
sum = foldr (+) 0
```

(a) Point-free

```
sum [] = 0
sum (x:xs) = x + sum xs
```

(b) Pointful

Figure 2.2: Point-free and pointful styles of a Haskell `sum` function

```
c = multiply(
  transpose(a, (1, 0)),
  expand_dims(b, 1))
```

(a) Point-free NumPy

```
c = for i j. a.j.i * b.j
```

(b) Pointful Dex

Figure 2.3: Point-free and pointful array programs in NumPy and Dex

2.2 Pointful array programming

In functional programming, one can distinguish **pointful** and **point-free** (tacit, or "pointless") styles – see examples in Figure 2.2. This distinction considers whether data flow is given by variable bindings, or driven with combinators. A classic example is that of λ and SKI calculi, which are respectively pointful and point-free. Both have the same expressive power, though compiling λ to SKI *(bracket abstraction)* incurs an overhead [23]. One can view the main result of this work as **bracket abstraction for array programs**.

We draw a similar distinction in array programming (following Paszke et al. [29]). The array programming model is said to be *point-free*, because we reason about operations on whole arrays at a time, and not individual elements. In contrast, in *pointful* (or *index-oriented*) array programming it is central to think about arrays as functions, with each element defined in terms of its index. The pointful style pushes indexing operations to the forefront, and brings us closer to mathematical notation – as per Iverson. We give a comparison between the *point-free* NumPy and *pointful* Dex in Figure 2.3.

2.2.1 Einstein summation

The need for a better notation for multidimensional operations became evident in the Python community, which looked towards **Einstein summation** as a solution. This notation is often used used in physics to express linear algebra in an index-oriented fashion. Indices which are repeated are implicitly summed over, and all indices span over the full size of the indexed axis [3]. A matrix product $C = AB$ is written:

$$C_{i,k} = A_{i,j} B_{j,k}$$

Einstein notation was the main influence on the `numpy.einsum` function, where the above is computed by `einsum("ij,jk->ik", a, b)`. This idea was expanded on by Tensor Comprehensions [35] to different reductions, and more recently in `einops` [31] to allow index-oriented shape manipulations, e.g.:

```
einops.rearrange(x, "b h w c -> b c h w ()") ⇝ expand_dims(transpose(x, (0, 3, 1, 2)), 4)
```

2.2.2 Languages

Dex is the most relevant example of a **pointful array language**, with its initial paper [29] introducing the *pointful/point-free* distinction for array programs. It features a value-dependent type system for keeping track of array sizes, embracing parallels between arrays and functions. However, I did not directly embed Dex in Python, as its dependent type system would likely be impractical in such a context, and it relies on a bespoke LLVM compiler.

```
def foo(x, y):
  return jax.lax.cond(
    x > 0,
    lambda y: y+1,
    lambda y: y-1,
    y
  )
```

(a) Deep-embedded DSL (JAX). The code is executed with tracing. The `cond` function is used, as `if` normally only works for Python `bool`s.

```
@function
def foo(x, y):
  if x > 0: z = y+1
  else:     z = y-1
  return z
```

(b) AST-based embedded DSL (TensorFlow). Python code is not executed, the `@function` decorator calculates a dataflow graph instead.

Figure 2.4: Domain-specific language embedding

One of the influences for Dex was $\tilde{F}$, introduced by Shaikhha et al. [33]. $\tilde{F}$ is used as a basis in a series of adjacent papers for researching array programming techniques. In this project it influenced my own pointful array calculus. However, $\tilde{F}$ does not have an open implementation, and its proposed compiler relies on C code generation, so this project takes a different approach to compilation.

The array comprehensions of Single-Assignment C can be seen as a precursor feature of pointful array programming [32]. Pointful array DSLs often feature comprehensions, which are prevalent in modern programming languages generally (Python, Haskell). I also took inspiration from the Futhark functional array language [18].

2.3 Domain-specific languages

The practice of creating domain-specific languages (DSLs) has a long history [19]. They are motivated by the observation that general capabilities like I/O, error handling, or even properties such as Turing-completeness are not always necessary language features. One can instead design languages tailored to a specific goal. Careful design choices simplify compilation and improve the programming experience.

2.3.1 Embeddings

A domain-specific language can be standalone, in which case it functions as an independent language with limited capabilities. However, a refined approach is **embedding** a DSL in a host language. In this project we focus on *deep-embedded*[2] DSLs that work on the basis of term constructors – they build up expressions in the DSL, and later execute them. In the context of Python, some DSLs directly inspect the program's AST instead of executing the code (compared in Figure 2.4). This style is inferior, as it is less predictable and cannot make good use of Python's features.

Generally, deeply embedded DSLs are much easier to integrate with existing codebases and features of the host language. In such DSLs **programs are values**, and hence the programmer can apply metaprogramming techniques to transform them [5]. A special case of an embedding is a *stringly-typed* DSL, where programs are expressed in strings which are parsed at runtime (such as the aforementioned `einops`). The problematic reality of such embeddings is that they are difficult to generalise beyond a small core.

[2] We further distinguish *shallow embeddings*, which do not build up an intermediate representation of the program [13] and hence cannot undergo whole-program optimisations. One can see PyTorch as an example. The project proposal confused deep and shallow embeddings, but it was clear from context a JAX-style deep embedding was intended.

```
def f(x, y):
    x2 = x * x
    return x2 + y
assert (f(Var('x'), Var('y'))
        == Add(Mul(Var('x'), Var('x')), Var('y')))
```

Figure 2.5: An annotated example of basic tracing with term constructors `Var`, `Add`, and `Mul`.

Tracing

In the context of Python, a common approach to creating deep-embedded DSLs is **tracing**. It has found use in JAX [11]. A DSL program is enclosed by a function in the host language, and to compile a program the function is *traced* by calling it with placeholder (symbolic) arguments, representing the variable inputs. Computations are performed *lazily* on these arguments – no work is done immediately, and instead a computational graph is constructed. Once the function returns, the graph is captured and the program can be compiled and executed. An example is given in Figure 2.5.

This approach leads to a kind of multi-stage programming, wherein before the results are determined, the entire program can first be collected and compiled. Tracing is often combined with techniques such as operator overloading, so that traced programs are written if they were *eager*. A limitation of this approach is that runtime-dependent control flow (e.g. `if`, `while`) cannot be traced directly.

2.4 Functional programming patterns

2.4.1 Applicative functors

McBride et al. [26] introduced the notion of an *applicative functor* – a functional programming pattern that generalises monads and specialises functors. It is a well-behaved structure that defines certain primitive operations, which behave well under function composition. One of the central structures introduced in this work (the Axial) is shown to be an applicative.

An applicative is formed by a *type constructor* f, meaning that for any type α there is some type $f\ \alpha$. We say a f is an applicative functor if the following operations are defined:

$$\begin{aligned} \circledast &: f\ (\alpha \to \beta) \to f\ \alpha \to f\ \beta \\ \mathrm{return} &: \alpha \to f\ \alpha \end{aligned}$$

These operations must follow the *applicative laws* (which we omit for brevity). These can be seen in the light of *homomorphisms* in a categorical sense, or just a notion of niceness under function composition. A definition upholding these laws shows the structure is *natural*, and thus in some way fundamental.

In this work we use a variation of $\circledast$ *(apply)*, called lift:

$$\begin{aligned} &\mathrm{lift} : (\alpha \to \beta \to \gamma) \to (f\ \alpha \to f\ \beta \to f\ \gamma) \\ &\mathrm{lift}\ f\ a\ b = (f \circledast a) \circledast b \end{aligned}$$

One further generalises lift to arbitrary arities lift_k via an analogous construction (the above lift is lift_2, and the base case lift_1 is a Functor map). There exist a bijection between constructions of lift_2 and $\circledast$.

Applicatives can be seen as an embellishment of values that permits application of similarly embellished transformations. Two examples of applicatives – List and ZipList – are shown in Figure 2.6.

$$\text{return}\ x = [x]$$
$$\text{lift}\ f\ a = \text{map}\ (\lambda\,(h, x).\ h\,x)\ (f \times a)$$

(a) List (nondeterminism) applicative for arbitrary lists – pairwise application ($\times$ is the Cartesian product for lists)

$$\text{return}\ x = \text{replicate}\ n\ x$$
$$\text{lift}\ f\ a = \text{map}\ (\lambda\,(h, x).\ h\,x)\ (\text{zip}\ f\ a)$$

(b) ZipList applicative on lists of fixed length n – respective application

Figure 2.6: Examples of two common applicative functors for lists

Representable (Naperian) functors are structures stronger than Applicatives, which generalise indexed collections. They are also useful abstraction in array programming Gibbons [12] and we follow their formulation to derive the Axial applicative.

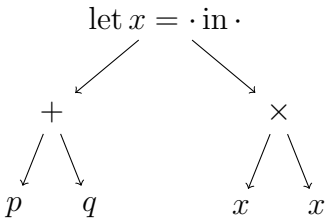
2.5 Compilers

Classically, a compiler consists of a lexer/parser (frontend), an assortment of analyses and tranformations (middle-end), and finally a code generator (backend), potentially with a runtime. In this project, the structure is slightly different. The frontend does not need a lexer/parser, as programs are constructed programmatically via the DSL embedding. We thus discuss the relevant program representations (Section 2.5.1) and compiler optimisations (Section 2.5.2).

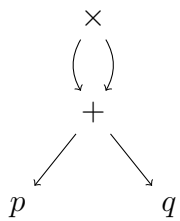
2.5.1 Program representations

A typical representation for programs is an **abstract syntax tree** (AST). Every node represents a single term constructor in the program, and child nodes are its direct subterms. Since the structure is a tree, every node has exactly one parent – except the root, which represents the entire program. In contrast, in a **term graph** there is no requirement that a node has at most one parent. Instead, the structure forms a directed acyclic graph,[3] where common subexpressions are represented with the same node. This is closely related to dataflow (computational) graphs, as repeated edges to a node correspond to data reuse. On the other hand, in an AST reuse is achieved explicitly through e.g. let-bindings.

In this work we deal both of these representations. Term graphs are more convenient for constructing the DSL program and some transformations. On the other hand, syntax trees are apt for staging the program's execution, as they make data reuse explicit. Examples of these can be seen in Figure 2.7.



(a) Abstract syntax tree



(b) Term graph (with the let-binding inlined)

Figure 2.7: Comparing representations of `let x = p + q in x * x`

[3] Unless there are infinite terms in the language.

2.5.2 Optimisations

Whenever a program contains multiple computations of a (pure) expression e, we may instead introduce a variable x defined to be e and replace all existing occurrences of e with x. This is dubbed **common subexpression elimination** (CSE). In the scope of this work, we consider a special case of CSE – transformation of a term graph into an abstract syntax tree by insertion of let-bindings.

Another important optimisation is **loop-invariant code motion**, which avoids loop computations where they are unnecessary. When an expression does not depend on the state of the loop it is placed in, it may be computed before the loop starts.

2.6 Requirements analysis

As outlined in the Introduction and the Proposal, the project contains the following core deliverables:

- **Formalisation** of a pointful array calculus, characterising what is expressible in the DSL.
- **Front-end embedding** of the DSL in Python, producing programs in the introduced calculus.
- **Execution back-end** for running programs efficiently by targeting existing array libraries.

There were many possible variations on any of these points, so best judgements were made to fulfil the success criteria. The calculus was originally inspired by Dex [29], with extensions influenced by $\tilde{F}$ [33]. A deep embedding was chosen for the DSL, as it is an elegant and flexible approach that easily lends itself to metaprogramming in the host language. I had to invent methods of executing a language based on array comprehensions with array libraries, as I was not aware of any existing approaches.

Since the paradigm does not possess particularly complex runtime features, a basic interpreter is easy to write. However, the goal of execution with the largest Python array programming library – NumPy – was prescribed, so the array calculus had to be adapted to the capabilities of the compilation scheme.

2.6.1 Methodology

Owing to the modularity of the project and relatively orthogonal extensions, the *spiral model* of development was adopted. After the initial milestones for each of the core deliverables (calculus, front-end, back-end), further ones focused on extensions. Priority was assigned based on impact on success criteria and practical usage. Risk assessment relied on the existence of implementations or descriptions of a feature in the literature. Debug tooling is particularly useful when building a compiler and was also given priority.

2.6.2 Review of array programs

Array programs vary significantly depending on the domain. One of these differences is in what language features are necessary to express them. For instance, deep learning programs rarely feature control flow, while differential equation solvers might perform iterated stencil computations. Even though application in deep learning is the original motivation, developing a more general calculus was preferable.

Due to my limited past experience with array programming applications, I conducted a comprehensive review of array programs. This assessment was conducted on 29 cases in the Futhark benchmark suite [15], with programs classified based on the language features they included.

An example conclusion from this review was the addition of a loop primitive (fold). It allows expressing more of the reviewed programs, which the original calculus could not (as it only had summation). Furthermore, a majority of the benchmark suite in the Evaluation is based on these cases. In addition to this review, I composed over 4000 words of design notes using *Obsidian* while creating Ein.

2.6.3 Choice of language and tools

Python was chosen as the main language, as it is the host language for the DSL, and has a robust array programming ecosystem. In particular, NumPy's first-class interface is in Python. Hence, the front-end and runtime were both fixed to Python. Only the middle/back-end could be moved to another language – and one with a robust Python cross-language interface. A notable example of such a language is Rust. However, attempts at an early prototype showed a trade-off due to its restrictive type system. Though it would have ensured better performance and reliability, my inexperience slowed down development. C++ is another Python-friendly alternative, but lacks language features useful for a compiler. Lastly, I considered OCaml, which is good for implementing compilers, but problematic with Python.

As Python is not the best choice for writing a compiler, a modern Python version was used to take advantage of structural pattern matching (added in version `3.10`) and optional typing. These significantly improved the code quality and programming experience. I chose a suite of tools standard to Python projects. All software applied was under a permissive open-source license, or allowed educational use.

2.6.4 Version control and testing

The main version control system applied for the project was Git. The repository was actively backed up to GitHub. All writing was done on-line in an Overleaf project synced to GitHub. I always held artefact copies on a local device. This was deemed sufficient due to high reliability standards of these services.

Python's `pytest` testing framework was applied. Project code was statically typed with Python's type annotation facilities, and checked by `mypy`. I also used a linter (`ruff`) and formatters (`black`, `isort`, `pyupgrade`). All of the tools were run via *pre-commit hooks*, ensuring a clean state of the repository at all times. Dependencies were managed using `poetry`.

2.7 Starting point

Prior to starting the project, I had a solid amount of experience in Python and NumPy – in particular through the *Scientific Computing* course. I had not designed or implemented a programming language before. I had done a fair amount of reading of primary sources on array languages to establish the feasibility of the project. Material learnt in *Semantics of Programming Languages* and *Compiler Construction* was useful throughout the course of the project.

No implementation code was written prior to Michaelmas 2024, though I experimented with some of the elements of the project. No existing codebases were used as a basis.

3 Implementation

We consider this project to be the implementation of a compiler for a domain-specific language. Throughout this chapter, we follow compiler phases from the frontend to the backend.

- We first introduce the *Phi calculus*, which formalises pointful array programs (Section 3.1).
- In Section 3.2 we showcase the embedded language – ***Ein***. Since we use an embedding, we do not need a classical parser. Ein's API builds up expressions in Phi, after which a complete program can be explicitly *evaluated*.
- To evaluate the program, we first have to *compile* it. We describe the most important analyses and transformations which form the compiler's middle-end in Sections 3.3 and 3.4.
- The key contribution of this work is the *code generation* scheme. I introduce the *Axial*, which forms a novel connection between pointful and point-free array programming (Section 3.5).
- We then define our compilation target for programs in the array programming model – *Yarr*, our point-free array calculus – and show how we compile Phi to Yarr through Axials (Section 3.6). A summary of the key compiler is given in
- Once the program is represented in Yarr, it is interpreted by an *execution backend* (runtime). We focus on the NumPy backend, but show the same approach also works for PyTorch and JAX (Section 3.7).
- We wrap up the chapter with a repository overview in Section 3.8.

The structure of Ein's compiler is depicted in Figure 3.1. Throughout this chapter code snippets are excerpts from executable Python using Ein.

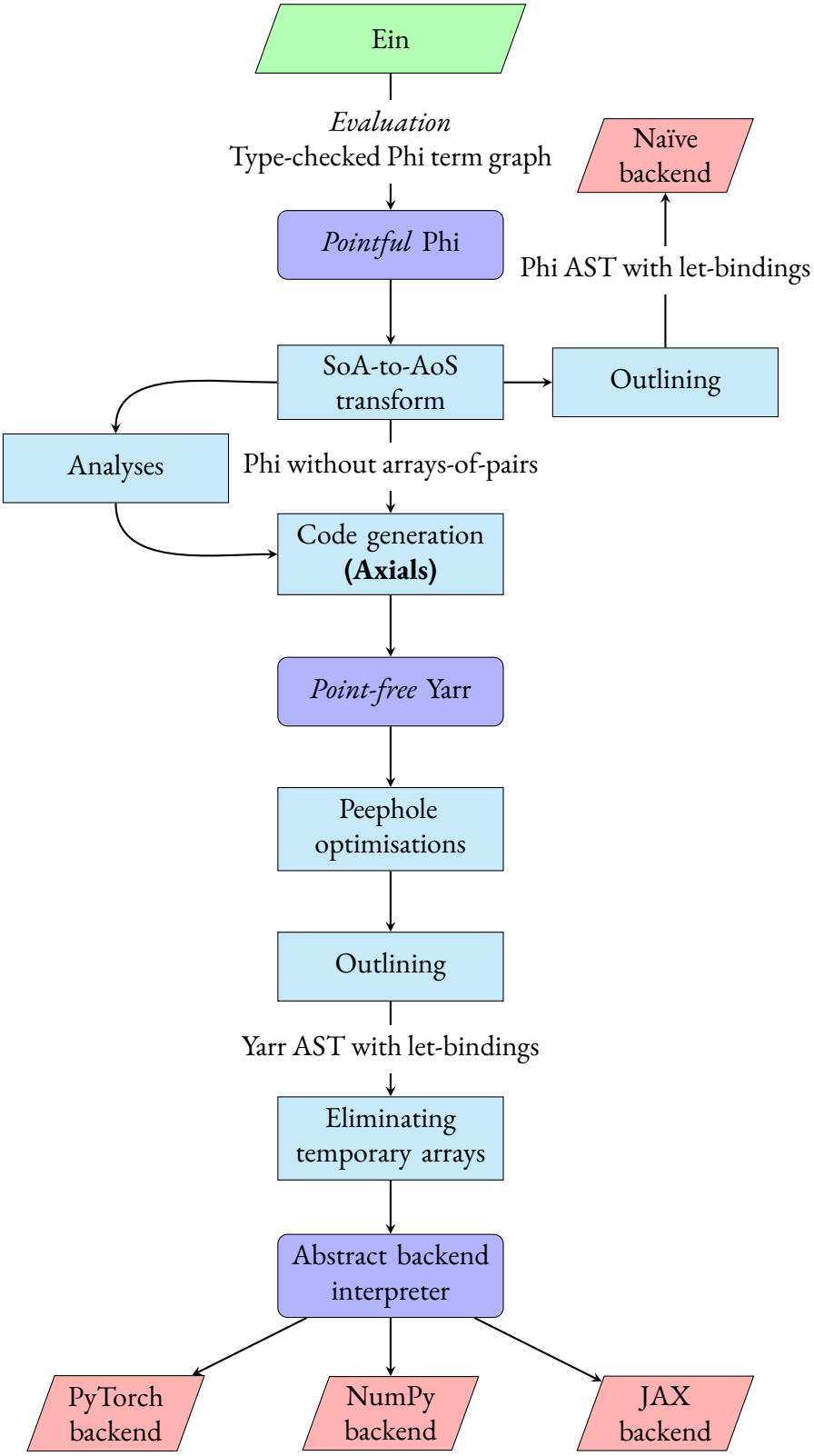


Figure 3.1: Structure of Ein's compiler. Intermediate languages are depicted in blue, compiler stages in light blue, and backends in red. Ein – the frontend DSL API, replacing a usual parser and lexer – is in green.

3.1 Theory – Phi calculus

We begin by introducing the functional **Phi calculus**, which is the theoretical basis for Ein. I outline the design choices made, and how they relate to the real capabilities of array libraries.

3.1.1 Syntax and design

We define the syntax and primitives of Phi. It is similar to $\tilde{F}$, as introduced by Shaikhha et al. [33].

$$
\begin{array}{rll}
e ::= & \Phi\, i[e].\, e \quad | \quad e[e] & \text{(array comprehension, indexing)} \\
| & \text{fold}\, x[e]\, \text{init}\, x = e\, \text{by}\, e & \text{(indexed fold)} \\
| & \langle e, e \rangle \quad | \quad \text{fst}\, e \quad | \quad \text{snd}\, e & \text{(pair construction, projections)} \\
| & \sigma(e, \ldots, e) & \text{(scalar operator)} \\
| & \text{assert}\, e = e \quad | \quad \text{size}_k\, e & \text{(equality assertion, size along axis } k \in \mathbb{N}\text{)} \\
| & \text{let}\, x = e\, \text{in}\, e & \text{(non-recursive let binding)} \\
| & x \quad | \quad i \quad | \quad c & \text{(variable, index, constant)}
\end{array}
$$

The introduction form for arrays is the indexed *array comprehension* Φ (pronounced *for*) – e.g. $v = \Phi\, i[5].\, i$ is the array $[0, 1, 2, 3, 4]$. The elimination form is *indexing* $a[i]$ (i-th element of a), so $v[2] = 2$. Phi interprets multidimensional arrays as either scalars (zero-dimensional base case) or vectors of arrays (successor case). In that respect indexing is into the *outermost* axis. For instance, $w = \Phi\, i[2].\, \Phi\, j[3].\, i + j = [[0, 1, 2], [1, 2, 3]]$, thus $w[1] = [1, 2, 3]$ and $w[1][2] = 3$.

The indexed fold facilitates a simple repeated iteration with an accumulator, and is closely related to the `loop` construct in Futhark. One can see Φ as perfectly parallel, while fold expresses sequential computation.

Examples of scalar operators σ include arithmetic $(+, \times, \ldots)$ and logic $(\wedge, \vee, \ldots)$ operators. Array sizes are obtained with the size primitive – if e has shape (n, m), then $\text{size}_1\, e = m$. We also include equality assertions to accommodate certain analyses (Section 3.3.2).

A crucial feature of Phi is the separation of identifiers into variables $(x, y, z, \ldots)$ and **indices** $(i, j, k, \ldots)$. Indices are solely introduced in array comprehensions, and receive special treatment in both the type system and the compilation scheme. We use usual variables $x, y, z, \ldots$ in all other cases.

For example, the following Phi term computes the (left-associative) sum $\sum_{i=0}^{n-1} a_i$ for a vector a:

$$\text{fold}\, i[n]\, \text{init}\, x = 0.0\, \text{by}\, x + a[i]$$

3.1.2 Type system

The type system of Phi is relatively straightforward, except for the handling of indices. Type constructors are unconstrained, and even allow arrays of pairs (missing from common Python array libraries).

$$
\begin{array}{ll}
\kappa ::= \text{Float} \mid \text{Int} \mid \text{Bool} & \text{(scalar types)} \\
\tau ::= \kappa \mid \square\tau \mid \tau \times \tau & \text{(Phi types – scalars, vectors, pairs)}
\end{array}
$$

The typing judgement $\Gamma; \Delta \vdash e : \tau$ is slightly non-standard due to the presence of indices. We use a separate environment for variables Γ and indices Δ. Consider the rules for array comprehensions and folds:

$$
\frac{\Gamma; \diamond \vdash e' : \text{Int} \qquad \Gamma; \Delta, i \vdash e : \tau}{\Gamma; \Delta \vdash \Phi\, i[e'].\, e : \square\tau}
\qquad
\frac{\Gamma; \diamond \vdash e' : \text{Int} \qquad \Gamma; \Delta \vdash a : \tau \qquad \Gamma, k : \text{Int}, x : \tau; \Delta \vdash e : \tau}{\Gamma; \Delta \vdash \text{fold}\, k[e']\, \text{init}\, x = a\, \text{by}\, e : \tau}
$$

Crucially, sizes and iteration counts are typed under an empty index environment $\Delta = \diamond$ – they cannot depend on any index. This ensures *regularity* of the parallelism involved. Since an (inner) array size cannot depend on any element's index, all arrays must remain rectangular. Similarly, since all iteration counts are the same across indices, the same computation is applied at each array element every iteration. Hence, jagged arrays like this one do not type:

$$\Phi\, i[5].\, \Phi\, j[i].\, i + j$$

Regularity is a beneficial property that ensures an efficient compilation scheme. We achieve it by a simple type check, which replaces a runtime check in Futhark, or the dependent type system in Dex. Other typing rules are relatively standard and carry through Γ and Δ.

3.1.3 Semantics

Conditionals Phi does not feature a dedicated conditional expression. We consider conditionals to be a ternary scalar operator instead, which we write $\mathrm{where}(c, t, f)$ (following the `numpy.where` primitive). As such, both branches are always evaluated regardless of the condition. This is a simple and convenient choice that aligns with what is implemented in array libraries like NumPy. Furthermore, it may lead to more efficient SIMD execution.

Out-of-bounds Since the ternary conditional 'where' evaluates both branches, even guarded indexing operations might end up out of bounds. JAX tackles a similar problem, and tends to clip indices or impute a default result. For simplicity, Phi clips indices into bounds.

3.1.4 Embedding Phi in Python

We embed Phi in Python to allow constructing and validating terms in the frontend. Conceptually, Phi's grammar is an ML-style sum type. To implement this pattern in Python, we use a sealed abstract base class `AbstractExpr` with a child class for each case of `Expr`. To construct the term $\Phi\, i[4].\, \Phi\, j[4].\, i \cdot j$, we write:

```
i, j = Index(), Index()
four = Const(Value(4))
table = Vec(i, four, Vec(j, four, Multiply((At(i), At(j)))))
```

To prevent the construction of invalid terms, constructors calculate and check the type (as in Figure 3.2).

```
class Vec(AbstractExpr):
    index: Index
    size: Expr
    body: Expr

    @cached_property
    def type(self) -> Type:
        if self.size.free_indices: raise TypeError(...)
        ...
        return VectorType(self.body.type)
```

Figure 3.2: An excerpt from the definition of `Vec` term, which corresponds to Phi's array comprehension Φ.

3.2 Frontend – Ein

Ein forms the programmer-accessible side of the project, and is implemented as the `ein` Python library. Unlike a traditional compiler, our frontend is not formed by a lexer or parser, but `ein`'s API. We overview Ein's features and its design as a purely functional, pointful array DSL.

3.2.1 Embedding

One of the main influences on Ein's embedding is JAX [11]. User-accessible Ein values encapsulate Phi expressions, and need to be explicitly *evaluated* by calling the `.eval()` method. Input NumPy arrays are provided with `wrap()`. As Ein is deep-embedded, *programming in Python is metaprogramming on Ein* [5]. Hence, Python functions act like Ein macros – and this is very useful for structuring Ein programs!

A convenient feature of Ein is that let-bindings are implicit, and they are instead inserted by the compiler. For example, say `e` is an Ein expression, so `e + e` indicates adding `e` to itself. At runtime, `e` is computed once.

3.2.2 Arrays

Similarly to Phi, Ein considers arrays to be defined recursively as either **scalars** or **vectors** of arrays. These correspond to the `Scalar` and `Vec[T]` classes. The methods of `Scalar` correspond to the scalar operators of Phi. These can be performed with operator overloading (as in `Scalar.__mul__` for `a * b`) and methods (`Scalar.sin` for `a.sin()`). Also, `Vec` implements `Vec.__getitem__`, so we can write `a[i]`.

```
a: Vec[Scalar]
b: Scalar = a[0].sin() * a[1].cos()
```

Both classes have an `expr` attribute which is used to build up corresponding Phi expressions, as in:

```
b.expr == Mul(
    Sin(Get(a.expr, const(0))),
    Cos(Get(a.expr, const(1))))
```

We further distinguish subclasses of `Scalar` corresponding to Phi scalars – `Int`, `Float` and `Bool`. At runtime, Ein represents `Int` with 64-bit signed integers, and `Float` with a double-precision floating point type.

3.2.3 Combinators

Pointful, comprehension-style *combinators* – `array` and `fold` – are central to Ein. Anonymous functions, defined with the Python `lambda` keyword, enable introduction of new variables – for instance, the index in an array comprehension. As such, `array(lambda i: i, size=5)` describes the Phi expression $\Phi\, i[5].\, i$. Further, the summation in let $a = \Phi\, i[5].\, i^2$ in $\sum_{i=0}^{4} a[i]$ is expressed with a `fold`:

```
a = array(lambda i: i*i, size=5)
s = fold(0, lambda i, acc: acc + a[i])
assert s.eval() == 0*0 + 1*1 + 2*2 + 3*3 + 4*4
```

We avoid explicit variable introductions by making use of the host language's lambda functions, in a manner similar to Atkey et al. [5] and JAX.[1] A longer Ein program is given in Figure 3.3.

We have noted NumPy is a first-order interface, as its operations cannot be parameterised by functions. On the other hand, `array` and `fold` are closer to Second-Order Array Combinators, which are indispensable in Futhark. Thus, combinators are what forms Ein's gain in expressive power versus NumPy.

[1] Explicit introductions are surprisingly common for their obvious drawback – there is nothing guarding against the reuse of variables in the wrong scope. Examples include Python's SymPy library or TACO. In the latter we have to write `i, j = pytaco.get_index_vars(2); S[i, j] = A[i, j] + A[j, i]` for `S = array(lambda i, j: A[i, j] + A[j, i])`.

```
def fold_sum(f):    return fold(0.0, lambda i, acc: acc + f(i))
def L1(u, v):       return fold_sum(lambda i: abs(u[i] - v[i]))
def pairwiseL1(A):  return array(lambda i, j: L1(A[i], A[j]))
```

Figure 3.3: Ein snippet computing the L_1 distances (sum of absolute differences) for each pair of rows of a matrix `A`. We can *separate concerns* across different Python functions in a simple, functional-esque way.

$$
\begin{array}{rcll}
\mu & ::= & \texttt{Int} \mid \texttt{Float} \mid \texttt{Bool} & \text{(scalars)} \\
& \mid & \texttt{Vec}[\mu] & \text{(vectors)} \\
& \mid & \texttt{dict}[\texttt{str}, \mu] \mid \texttt{tuple}[\mu, \ldots] \mid \texttt{Dataclass}_{\mu} & \text{(records)}
\end{array}
$$

Figure 3.4: Types μ under which Ein's primitives are *closed*, mirroring the `mypy`-style type annotations in Section 3.2.7. Dataclasses (similar to Java's records) are assumed to only have fields of μ. Indexing into a vector of records returns the actual Python container – e.g. indexing `Vec[tuple[Int, Float]]` yields `tuple[Int, Float]`.

3.2.4 Size inference

Size inference allows omitting the size of arrays in some contexts. Consider the following computation:

```
array(lambda i: a[i] + b[i], size=a.size(axis=0))
```

Say that the vectors `a` and `b` have the same size. Then, with size inference, we may omit the explicit `size`:

```
array(lambda i: a[i] + b[i])
```

Specifically, for any index `i` that does not have an explicit `size` defined, it is inferred by taking the size of any array `a` that is indexed directly with `i` (i.e. in an expression `a[i]`). Where there are other such candidates `b`, we add an assertion term that `a` and `b` have the same size. We further generalise this to `fold`:

```
fold(0.0, lambda i, acc: 0.1 * a[i] + 0.9 * acc)
```

There are many mechanisms similar to size inference, such as Dex's inference of dependent (index) types.

3.2.5 Records

Ein uses Phi's pair types to represent *record types*. We use a basic encoding: $\{x : \text{Int}, y : \text{Float}, z : \square\text{Int}\}$ becomes the Phi type $\text{Int} \times (\text{Float} \times \square\text{Int})$. In contrast to Ein's `Vec` and `Scalar`, we do not use a custom Ein class for representing records – instead, we use Python's builtin container types. We summarise the types μ accepted and returned by Ein's API in Figure 3.4. An illustrative example of records is given below:

```
# Array of dictionaries (records {x: int, y: int, z: int})
a = array(lambda i: {"x": i, "y": i*i, "z": i*i*i}, size=10)
# Indexing into a returns a dictionary with the same keys (record fields)
assert list(a[4].keys()) == ["x", "y", "z"]
assert a[4]["y"].eval() == 16
```

Records enable a style of array programming unavailable in established Python array libraries. It is possible to describe composable array structures of custom types and define operations on them. For instance, one could define an array of dual numbers[2] with overloaded operators by using a dataclass:

[2] Dual numbers are similar to complex numbers, but instead of the imaginary unit $i^2 = -1$ we instead have a symbol $\varepsilon^2 = 0$. They are particularly useful in forward-mode automatic differentiation, and one could use them for this purpose in Ein.

```
@dataclass
class Dual:
    real: Float
    eps: Float
    def __mul__(self, other: Dual) -> Dual:
        return Dual(self.real * other.real,
                    self.real * other.eps + self.eps * other.real)
a = array(lambda i: Dual(i, 1.0), size=5)  # constructs Vec[Dual]
b = array(lambda i: a[i] * a[i])           # calls Dual.__mul__
```

In contrast, most renditions of the array programming model struggle to achieve this sort of composability, as they tend to just consider *whole arrays of primitives*. In Python, at best one could sub-class an array type, but this would not compose as well. With records, we can define overloaded operations and reuse existing code through **duck typing**. Efficient representation of records is ensured via program transformations (Section 3.4.2), and they are a zero-cost abstraction.

3.2.6 Reductions

Ein also possesses a `reduce` combinator. It can be thought of as a `fold` for *associative* operations, i.e. $\circ$ such that $x \circ (y \circ z) = (x \circ y) \circ z$. For instance, the following expresses a summation of `v: Vec[Float]`:

```
v.reduce(0.0, lambda x, y: x + y)
```

In contrast to the limited set of specialised reductions offered in NumPy, Ein's reductions work for arbitrary operators. At runtime, we rely on a work-efficient scheme on a binary tree, taking advantage of data parallelism. This can be orders of magnitude faster than a `fold` based on an inefficient Python loop. Array comprehensions together with reductions allow expressing *list homomorphisms* (map-reduce) [8]. One can implement `argmin` as a reduction over records $\{\text{val} : \text{Float}, \text{idx} : \text{Int}\}$ like so:

```
array(lambda i: {"val": a[i], "idx": i}).reduce(
  {"val": float("inf"), "idx": 0},
  lambda x, y: where(x["val"] < y["val"], x, y)
)["idx"]
```

Since reductions are a special case of a fold (`v.reduce(z, f)` $\rightsquigarrow$ `fold(z, lambda i, x: f(x, v[i]))`), there are few differences in the compiler, and we do not elaborate on them much.

3.2.7 Type annotations

Ein classes are usable as Python type annotations. For instance, `a: Vec[Vec[Float]]` indicates `a` is an Ein matrix of floats. Thanks to this design, it is possible to use standard type checkers like `mypy` for **optional typing** of Python programs using Ein. Some type errors – like attempting to add a scalar and a vector, or indexing into a scalar – may be discovered before runtime. Furthermore, type hints for records work on the same principle – we previously wrote `v: Vec[Dual]` for a vector `v` of dual numbers. Indexing into `v` returns `v[i]: Dual`, which is known to have the fields `v[i].real: Float` and `v[i].eps: Float`.

Signatures of combinators in Ein

```
def array(f: (Int) -> T, size: Int | None = None) -> Vec[T]: ...
def fold(init: T, step: (Int, T) -> T, count: Int | None = None) -> T: ...
def reduce(vec: Vec[T], ident: T, cat: (T, T) -> T) -> T: ...
```

$$
\begin{aligned}
(\!| i |\!) &= 1 \times i + 0 && \\
(\!| c \times e |\!) &= (c \times a) \times i + (c \times b) && \text{if } (\!| e |\!) = a \times i + b \\
(\!| e + e' |\!) &= (a + a') \times i + (b + b') && \text{if } (\!| e |\!) = a \times i + b \\
& && \text{and } (\!| e' |\!) = a' \times i + b'
\end{aligned}
$$

Figure 3.5: We write $(\!| e |\!) = a \times i + b$ for the affine normal form of e (if it exists). If an expression can be decomposed through matching on these cases, then we say it has the normal form.

3.3 Analyses

We describe the main analyses applied on Phi, which facilitate further transformations and code generation.

3.3.1 Normalisation of high-level operations

Phi does not perfectly correspond to NumPy routines. For instance, to express a summation of a one might write fold $i[\text{size}_0\, a]$ init $x = 0.0$ by $x + a[i]$, while in NumPy a `sum` routine is available. Thus, it becomes reasonable to match on *high-level operations* within Phi programs that are semantically equivalent to some NumPy routine. However, a basic approach is hardly sufficient. For instance, $a[\max(i-1, 0)]$ could be efficiently interpreted as a variation of:

```
numpy.pad(
    a[:-1],               # skip final element
    (1, 0), mode='edge'   # pad 1 at start with first value
)
```

How would a compiler *synthesise* this slicing and padding? This is impossible in general without a formal specification of each NumPy routine. Hence, I only settle for solving common cases. The general technique applied is a variation of *normalisation by evaluation*. Phi terms are reflected into normal forms, which we can **compose** under a certain algebraic structure. Later, these can be reified into our array program representation (Yarr; Section 3.6.1).

For example, consider indexing of the form $\Phi\, i[n].\, x[a \times i + b]$, where (x, a, b) are expressions with no free indices. This essentially[3] corresponds to the slicing `x[b::a]` – a subarray starting at index `b` and stepping by `a`. The simplest approach would be to match on the syntax $a \times i + b$, but this is unstable with respect to small changes to the program (like $b + a \times i$, or $(a \times i + b) + b'$). To resolve this, we appeal to the semantics of Phi and the algebraic structure of affine functions (see Figure 3.5). If for some $e[e']$ the item e' has an *affine normal form* $a \times i + b$, we can synthesise a slicing.

We generalise the normalisation to the following forms:

- 'Clipped-shifts', which correspond to slicing and padding. The normal form is $\min(\max(i+c, l), u)$.
- Sums of generalised Einstein summations (tensor contractions). Thanks to linearity, these have an algebraic structure closed under addition, multiplication, and summation. Using this form, we can generate calls to matrix multiplication routines (through `numpy.einsum`). This could be generalised to semirings other than $(+, \times)$, in which case we could e.g. call GraphBLAS.

[3] There are some caveats when indices go out of bounds. Nevertheless, let us presume it is easy to synthesise the right operation.

$$\text{size}_0\ \Phi\ i[e'].\ e \equiv e' \quad \text{size}_{k+1}\ \Phi\ i[\cdot].\ e \equiv \text{size}_k\ e \quad \text{size}_k\ e[e'] \equiv \text{size}_{k+1}\ e \quad (\text{assert}\ e = e') \equiv e \equiv e'$$

$$\frac{\text{size}_k\ e \equiv \text{size}_k\ e' \equiv e''}{\text{size}_k\ (\text{fold}\ i[\cdot]\ \text{init}\ x = e\ \text{by}\ e') \equiv e''}$$

Figure 3.6: Example rules of the Phi term equivalence judgement $\equiv$. The last rule states that if the size of an accumulator e is the same at the start and after an iteration $e \mapsto e'$, the final result also has the same size.

3.3.2 Size equivalences

To apply some optimisations, we need to reason about sizes of arrays. We do this by introducing a simple *term equivalence judgement* $\equiv$ (as in some dependent type systems), which allows safe approximation of when array-valued expressions must yield the same sizes. Example rules are given in Figure 3.6.

In the implementation, we traverse the entire program term graph, matching on the rules and unifying equivalence classes on a disjoint-set data structure [9]. This could be expanded to use an e-graph structure.

3.4 Transformations

3.4.1 Outlining

The term graph form produced by Ein is not an intermediate representation apt for generating executable code – it lacks information on *when* values should be computed. To address this, we *outline* the **term graph** into an **abstract syntax tree** of linear size. *Inlining* is the removal of the let bindings inserted by outlining.

The two main techniques applied are common subexpression elimination and loop-invariant code motion. The former rewrites computations like $e + e$ to $\text{let}\ x = e\ \text{in}\ x + x$. Meanwhile, the latter ensures that terms constant across loop iterations are computed before the loop, as in:

$$\Phi\ i[n].\ a[i] + (\Phi\ j[n].\ 2 \cdot b[j])[i] \quad \rightsquigarrow \quad \text{let}\ c = \Phi\ i[n].\ 2 \cdot b[i]\ \text{in}\ \Phi\ i[n].\ a[i] + c[i]$$

We consider both of these transformations to be a special cases of **let insertion**. For simplicity, we associate the possible locations with nodes in the term graph. Thus for each term t we find a list of terms t' to bind at t – meaning we rewrite $t \rightsquigarrow \text{let}\ x = t'\ \text{in}\ \{x/t'\}t$. Here, $\{x/t'\}$ replaces the term t' with x.

An important simplification of the problem setup is that Phi calculus is pure. We assume all loops iterate at least once (i.e. Φ and fold), thus ensuring all subexpressions are evaluated at least once.

Common subexpression elimination We perform CSE via a standard method on term graphs, which is by application of **dominators**. Consider a flow graph in which there is an edge $t \to t'$ whenever t' is a direct subterm of t, with the source set to the entire term. Then consider the immediate dominator t^* of a term t – t^* is the smallest term that contains all instances of t (Figure 3.7 (a)). Hence, we should let-bind t at t^*. We perform this insertion for every non-atomic term t which has more than one parent.

We compute dominators with the `networkx` package, which has $O(n^2)$ time complexity in the worst-case. Since our graphs are acyclic, we could use an alternative $O(n \log n)$ algorithm [30].

Loop-invariant code motion To compute a loop-invariant value prior to the loop, we need to determine whether it is unaffected by the loop state. The approach for Phi relies on a tree traversal keeping track of a stack of binders (of loop state variables). A computation can be inserted at the outermost point where it does not invalidate lexical scope constraints (Figure 3.7 (b)).

(a) The entire term $(u+v)+((u+v)\times w$ is the immediate dominator of $u+v$.

(b) $f[i]$ cannot be moved outside $\Phi\, i[n].\, e + f[i]$, as it would *skip* the binder of i ($\Phi i[n]$) – e can be.

Figure 3.7: Example computations in common subexpression elimination (a) and loop-invariant code motion (b)

A tricky part of outlining is the ordering in which let-insertions are performed. Making the wrong choices results in scope errors or infinite terms. There are cases in which bindings may become trivial (e.g. $\text{let}\, x = y\, \text{in}\, e$). To erase these, we afterwards perform **copy propagation** by inlining all trivial bindings.

3.4.2 Array-of-structs to struct-of-arrays

A standard technique in compiling array programs is the array-of-structs (AoS) to struct-of-arrays (SoA) transformation [33]. The Ein compiler uses it to reduce all values to simple arrays of primitive types. This is crucial, as our compilation targets lack support for arrays of composite types like pairs.[4] Once we eliminate arrays of pairs, the only types τ' are given by:

$$\alpha ::= \kappa \mid \square\alpha$$
$$\tau' ::= \alpha \mid \alpha \times \tau'$$

Hence, τ' can only be tuples of arrays. This simplifies compilation and runtime value representation.

The transformation is depicted in Figure 3.8. It relies on the isomorphism $\square(\tau_1 \times \tau_2) \cong \square\tau_1 \times \square\tau_2$, i.e. for each array of pairs there is an equivalent pair of arrays of equal size.

$$\mathcal{S}[\![\Phi\, i[e_n].\, e]\!] = \langle \Phi\, i'[\mathcal{S}[\![e_n]\!]].\, \text{fst}\, \mathcal{S}[\![e]\!], \Phi\, i''[\mathcal{S}[\![e_n]\!]].\, \text{snd}\, \mathcal{S}[\![e]\!] \rangle \quad (\text{where } \Gamma; \Delta, i \vdash e : \tau_1 \times \tau_2)$$
$$\mathcal{P}[\![e, e']\!] = \langle \mathcal{P}[\![\text{fst}\, e, e']\!], \mathcal{P}[\![\text{snd}\, e, e']\!] \rangle \quad (\text{where } \Gamma; \Delta \vdash e : \tau_1 \times \tau_2)$$
$$\mathcal{P}[\![e, e']\!] = e[e'] \quad (\text{where } \Gamma; \Delta \vdash e : \square\tau)$$
$$\mathcal{S}[\![e[e']]\!] = \mathcal{P}[\![\mathcal{S}[\![e]\!], \mathcal{S}[\![e']\!]]\!] \quad (\text{where } \Gamma; \Delta \vdash e : \square\tau)$$
$$\mathcal{S}[\![e + e']\!] = \mathcal{S}[\![e]\!] + \mathcal{S}[\![e']\!] \quad (\text{and similarly for other cases})$$
$$\mathcal{S}[\![\square(\tau_1 \times \tau_2)]\!] = \square\tau_1 \times \square\tau_2 \quad (\text{identity otherwise})$$

$$\text{fst}\, \langle x, - \rangle \rightsquigarrow x \quad \text{snd}\, \langle -, y \rangle \rightsquigarrow y \quad \langle \text{fst}\, p, \text{snd}\, p \rangle \rightsquigarrow p$$

Figure 3.8: The core of the type-driven AoS-to-SoA transformation on Phi, denoted by $\mathcal{S}$. $\mathcal{P}$ maps an indexing $e[e']$ to all arrays in the tuple representation of e. Environments $\Gamma; \Delta$ are passed through implicitly, in accordance with typing rules. It is succeeded by peephole optimisations which eliminate redundant pairs.

[4] Technically, NumPy does allow defining custom `dtypes`, but this feature is mostly intended for (de)serialising binary data. Similar features are nevertheless unavailable in most other array libraries.

3.5 Escaping the Pointless with Axials

Axials are the key contribution of this work. They establish a new formal connection from pointful to point-free array programming. Thus, we can escape programming pointless (point-free) code by translating pointful code to it. Their structure is natural, in the sense that it forms an applicative functor.

Motivation We shall use $A \simeq B$ to denote an isomorphism, where there exist a bijection between sets A and B. To motivate Axials, we appeal to the separation of identifiers in Phi into variables $x, y, z, \ldots$ and indices $i, j, k, \ldots$. Hence, there is a similar separation of environments $\text{Env} \equiv (\text{Var} + \text{Index}) \to \text{Value}$:

$$\text{Env} \simeq \text{Var} \times \text{IndexEnv}$$

Let us now consider the denotation of a Phi expression e, given by $[\![e]\!] : \text{Env} \to \text{Value}$. We can now apply currying within the denotation:

$$[\![e]\!] : \text{Env} \to \text{Value} \simeq \text{VarEnv} \times \text{IndexEnv} \to \text{Value} \simeq \text{VarEnv} \to \boxed{\text{IndexEnv} \to \text{Value}}$$

Functions $\text{IndexEnv} \to \text{Value}$ turn out to have a lot of structure. Since indices are only introduced in comprehensions, we know their values must be within a range of integers like $i \in [0, s_i)$ – where s_i is the *size* of an index i, as introduced in $\Phi\, i[s_i].\, e$. This leads us to say that:

$$\text{dom}\, \text{IndexEnv} \simeq [0, s_i) \times [0, s_j) \times [0, s_k) \times \ldots$$

where $\iota = \{i, j, k, \ldots\}$ are the free indices in e. Indeed, this *index space* directly corresponds to that of a $|\iota|$-dimensional array of shape $(s_i, s_j, s_k, \ldots)$ – the set of which we denote $\text{Array}_{|\iota|}$. The difference is that our 'array' has *labelled* axes, rather than *positional* ones. We denote $\iota!$ as the set of permutations of ι.[5] To complete the connection with arrays, we need to order our labelled axes:

$$\text{IndexEnv} \to \text{Value} \simeq \iota! \to \text{Array}_{|\iota|}$$

We have obtained that expressions of type τ dependent on indices ι can be evaluated as a $|\iota|$-dimensional array of τ. We shall assume onwards indices are *intrinsically sized*, meaning we have a constant $s_i \in \mathbb{N}$.[6]

We can finally substitute the result back into the type of $[\![e]\!]$:

$$\boxed{[\![e]\!] : \text{Env} \to \text{Value} \simeq \text{VarEnv} \to \iota! \to \text{Array}_{|\iota|}}$$

We took the jump from pointful *(given an assignment to each index, compute τ...)* to point-free style *(using an array of values τ at each possible index...)*. **Axials** are an encoding of $\iota! \times \text{Array}_{|\iota|}$, fixing the ordered *axes* and encapsulating the connection on the point-free side. We define this encoding so that it is natural, and thus composes well from subexpressions of e.

[5] Throughout this text permutations are *orderings* of a set – sequences where each element occurs exactly once. If $\iota = \{i, j, k\}$, then $\iota! = \{[i, j, k], [i, k, j], [j, i, k], [j, k, i], [k, i, j], [k, j, i]\}$.

[6] Assuming all sizes are program constants is a useful assumption to simplify the reasoning. In general, there is a **unique** Phi *expression* which computes the size of an index's range – and this is what the compilation scheme relies on.

Definition We define an Axial to be the type constructor $\mathrm{Axial}\,\alpha \equiv \mathrm{List}\,\mathrm{Index} \times \mathrm{Array}\,\alpha$. Elements of the list are called the *axes* (Index is a type representing indices i; axes correspond to $\iota!$ above), while the array is called the *value* of the axial. We assume the rank of the value is the same as the number of axes, and each dimension of the value has the same size as the axis to which it corresponds.

We show Axials form an applicative functor. To this end, we need to define:[7]

$$\begin{aligned}\mathrm{pure} &: \alpha \to \mathrm{Axial}\,\alpha \\ \mathrm{lift} &: (\alpha \to \beta \to \gamma) \to (\mathrm{Axial}\,\alpha \to \mathrm{Axial}\,\beta \to \mathrm{Axial}\,\gamma)\end{aligned}$$

The former, pure, is straightforward – we can simply write:

$$\mathrm{pure}\,a = \langle [], a \rangle$$

On the other hand, to define $\mathrm{lift}\ f\ a\ b$ we need to think of an axial as a collection. We follow the formulation for ZipList (or Naperian functors [12] more generally) applying f *elementwise* to a and b:

$$\mathrm{lift}\ f\ a\ b \sim \Phi\, i.\ f(a[i], b[i])$$

We need to perform a similar operation in a *axis-aware* manner. The idea may be illustrated as follows. Say we have axials $a = \langle [i, j], \mathsf{a} \rangle$ and $b = \langle [k, j], \mathsf{b} \rangle$. We want to take an ordered union of their axes (free indices), producing an Axial $c = \langle \gamma, \mathsf{c} \rangle$. We might *choose* the axes $\gamma = [i, j, k]$, in which case c is given by:

$$\mathsf{c}[i, j, k] = f(\mathsf{a}[i, j], \mathsf{b}[k, j])$$

this is essentially an instance of **broadcasting**, solidifying a connection with point-free array programming.

Without going into detail (the Axial setup is a bit different in the compilation scheme) and with some abuse of notation, we define the following lift for axials $a = \langle \alpha, \mathsf{a} \rangle$ and $b = \langle \beta, \mathsf{b} \rangle$:

$$\mathrm{lift}\ f\ a\ b = \left\langle \gamma, \Phi\, \overline{\gamma}.f\left(\mathsf{a}[\overline{\gamma}]^{\gamma}_{\alpha}, \mathsf{b}[\overline{\gamma}]^{\gamma}_{\beta}\right) \right\rangle \quad \text{where } \gamma = \alpha \sqcup \beta$$

where $\gamma = \alpha \sqcup \beta$ deterministically picks some permutation of axes from both α and β, and $\Phi\, \overline{\gamma}$ constructs a multidimensional array with indices in γ (and sizes consistent with a and b). Indexing into a and b respects the new axes γ, removing and permuting indices.

Context Why are Axials useful in relating different styles of array programming? Crucially, instances of lift are essentially NumPy-style **broadcasting**. In fact, Axials can be seen as a generalisation of arrays. Instead of *positional axes*, we have a notion of *labelled axes*. This is reminiscent of the recently introduced *named tensors* [7].

It turns out Axials relate to two well-known applicative functors – the List and ZipList. So far, we did not specify Index beyond assuming it has a notion of equality. Under the right Index, we obtain (roughly) the behaviour of existing applicatives. When we take $\mathrm{Index} \equiv \mathrm{Unit}$ we obtain the ZipList, for which lift always operates elementwise. It is a well-known fact the ZipList cannot be generalised to a monad, thus there is no obvious Axial monad. On the other hand, when $\mathrm{Index} \equiv \mathrm{Bool}$ we obtain the List applicative, for which lift operates on the Cartesian product.

[7] A complete proof would also show the definitions satisfy the *applicative laws* (i.e. behave well under function composition).

3.6 Code generation – from pointful to point-free

We now finally talk about how we can make the step from the pointful Phi calculus into the point-free array programming model. We devise *Yarr* (a point-free array calculus), which is our target program representation. We show how Phi programs can be *lifted* with the Axial, yielding a simple compilation scheme.

3.6.1 Compilation target – Yarr, the array calculus

We use a straightforward formalisation of point-free array programs for our compilation target, abstracting the interface of NumPy. Figure 3.9 shows the basic formulation into which we can compile Phi. It is extended with reductions, concatenation, specialised indexing operations (e.g. slicing), padding, and generalised Einstein summations. Note that the only significant common part with Phi is the indexed fold. In particular, indices i from array comprehensions are erased entirely. The compilation scheme guarantees that no comprehension results in a fold. Otherwise, compiling Phi would be much easier – one could set array elements one-by-one.

$$
\begin{array}{rll}
\tau' ::= & \kappa \mid \square\tau' & \text{(arrays)} \\
\tau ::= & \tau' \quad | \quad \tau' \times \tau' \quad | \quad \dots & \text{(tuples of arrays)} \\
E ::= & \mathrm{range}(E) & \text{(naturals up to } n\text{)} \\
| & \mathrm{size}(E, n) & \text{(array size along axis)} \\
| & \mathrm{transpose}(E, (n, \dots)) & \text{(permute axes)} \\
| & \mathrm{squeeze}(E, n) & \text{(remove 1-axis)} \\
| & \mathrm{unsqueeze}(E, n) & \text{(add 1-axis)} \\
| & \mathrm{repeat}(E, E, n) & \text{(repeat along axis)} \\
| & \mathrm{gather}(E, E, n) & \text{(indexing along axis)} \\
| & \mathrm{elementwise}_\sigma(E, \dots) & \text{(elementwise op.)} \\
| & \langle E, \dots \rangle \quad | \quad \mathrm{proj}_n(E) & \text{(tuples and projections)} \\
| & \mathrm{let}\, x = E \,\mathrm{in}\, E & \text{(let-binding)} \\
| & \mathrm{fold}\, x[E] \,\mathrm{init}\, x = E \,\mathrm{by}\, E & \text{(indexed fold)} \\
| & x \quad | \quad c & \text{(variables, constants)}
\end{array}
$$

Figure 3.9: The core of Yarr, our point-free array calculus. Operations are based on core NumPy primitives.

3.6.2 Compilation scheme

We now describe our scheme for compiling Phi into Yarr. It could be said it *lifts* Phi expressions into the Axial applicative, at which point they are reified to computations in Yarr.

We introduce a transformation $\mathcal{A}[\![e]\!]$ mapping Phi terms e to an Axial of Yarr terms E:

$$\mathcal{A}[\![e]\!] = \langle \pi, E \rangle$$

where π is a permutation of the free indices $\iota(e) \subseteq \mathrm{Index}$ of e, called *axes* (as in Axials). The encoding of $\mathcal{A}$ 'flattens' the Axial *values*, so $\mathrm{rank}(E) = |\pi| + \mathrm{rank}(e)$. Thanks to Axials forming an applicative, $\mathcal{A}[\![e]\!]$ can be composed from $\mathcal{A}$ for subterms of e. For a closed Phi program p, we expect that P s.t. $\mathcal{A}[\![p]\!] = \langle [], P \rangle$ is its Yarr equivalent. It remains to fill in the gap where there are free indices (i.e. $\pi = \left[\pi^{(1)}, \ldots, \pi^{(k)}\right]$ is non-empty). We require the following *semantic equivalence* property for $\mathcal{A}[\![e]\!] = \langle \pi, E \rangle$:

$$[\![\Phi\, \pi^{(1)}[s_{\pi^{(1)}}]. \ldots \Phi\, \pi^{(k)}[s_{\pi^{(k)}}].\, e]\!] = [\![E]\!]$$

where s_i (in general, a Phi term) is used to denote the size of the range of index i. For example, consider $p = \Phi\, i[2].\, \Phi\, j[3].\, i + j$, so that $s_i = 2$ and $s_j = 3$. Then the following satisfies the property at $e \mapsto i + j$:

$$\mathcal{A}[\![i + j]\!] = \langle [i, j], [[0, 1, 2], [1, 2, 3]] \rangle$$

We can now proceed to describe $\mathcal{A}$ for various Phi constructs. Firstly, we assumed that for any index i, we know the size s_i. We achieve this by adding auxiliary Phi terms $i \triangleleft s_i$.[8] Therefore:

$$\mathcal{A}[\![i \triangleleft s_i]\!] = \langle [i], \mathrm{range}(S_i) \rangle \quad \text{where } \mathcal{A}[\![s_i]\!] = \langle [], S_i \rangle$$

To construct these auxiliaries, we substitute them at comprehensions (which introduce indices). Outside the comprehension body the index i is no longer free, so we remove i from axes and adjust the encoding:

$$\mathcal{A}[\![\Phi\, i[s_i].\, e]\!] = \left\langle \pi \setminus\!\setminus i, \mathrm{transpose}\left(E, \theta^i_\pi\right) \right\rangle \quad \text{where } \mathcal{A}[\![\{i \triangleleft s_i / i\} e]\!] = \langle \pi, E \rangle$$

where $\pi \setminus\!\setminus i$ removes i from π, and θ^i_π is the permutation that transposes the index i onto the position of the outermost (Phi) axis. If i was not free in e, we would use an appropriate repeat instead.

We now get to apply the Axial applicative. Constants c are handled by $\mathcal{A}[\![c]\!] = \mathrm{pure}\, c$, and thus they have no axes. The other defining operation of the Axial applicative – lift – also plays a central role. Take a scalar operator $e = \sigma(e_1, e_2)$, with $\mathcal{A}[\![e_1]\!] = \langle \pi_1, E_1 \rangle$ and $\mathcal{A}[\![e_2]\!] = \langle \pi_2, E_2 \rangle$. We want to *align* the results of E_1 and E_2 so that they have compatible axes π which we can **broadcast** over. Here, π needs to be a permutation of the indices $\iota(e_1) \cup \iota(e_2)$. This new choice of axes requires coercing E_p ($p \in \{1, 2\}$) – we need to first transpose E_p (to match the order in π from π_p), and then unsqueeze missing axes (the ones missing in π_p). We write $\pi = \mathrm{aligned}(\pi_1, \pi_2)$ for choosing new axes, and the change-of-axes $\mathrm{align}(E_p, \pi_p, \pi)$. The final result is essentially an Axial lift σ:

$$\begin{aligned}
&\pi = \mathrm{aligned}(\pi_1, \pi_2) \\
&E'_1 = \mathrm{align}(E_1, \pi_1, \pi) \quad E'_2 = \mathrm{align}(E_2, \pi_2, \pi) \\
&\mathcal{A}[\![\sigma(e_1, e_2)]\!] = \langle \pi, \mathrm{elementwise}_\sigma(E'_1, E'_2) \rangle
\end{aligned}$$

[8] In the implementation, it is more convenient to pass through a context instead.

We have covered the general technique on the core Phi constructs. We briefly overview the rest:

Indexing Indexing $e = e_1[e_2]$ is done in a very similar manner to elementwise operations, instead synthesising a gather operator. Shape manipulation gets more involved – general indexing is notoriously difficult to express in NumPy. Furthermore, there are many kinds of specialised indexing routines. Besides ones in Section 3.3.1 we have e.g. `numpy.take`, applied when e_1 and e_2 have disjoint axes.

Folds Folds are an interesting case. Both the accumulator and the body have axes which need to be aligned. Since iteration counts are independent of indices i, the fold step is perfectly data-parallel. Due to fold's generality, at runtime we need to interpret it as a `for` loop – Python's overhead is helped here by data parallelism.

Tuples After the SoA-to-AoS transform (Section 3.4.2), we need only consider tuples of arrays of primitives. The encoding is generalised to support them by prepending axes π to each array in the tuple.

The Axial encoding is not unique due to the choice of *axes* by the align function. This choice has an impact on performance, as it corresponds to the in-memory representation of the array. We use a simple deterministic heuristic to minimise the transpositions needed.

3.7 Runtime – Execution backends

The compilation target of Ein is a high-level library in the array programming model. This means that most of the work is spent in large-workload whole-array operations, dominating the many overheads associated with Python. As such, we take some liberties to simplify the implementation. The primary technique we use is a *staging* function

$$\text{Expr} \to (\text{Env} \to \text{Value})$$

where Expr is an expression in Phi or Yarr, and Env is a mapping from variables to assigned values. This essentially implements the denotational semantics $[\![e]\!] : \text{Env} \to \text{Value}$, composing an interpreter from denotations of each subexpression e. We represent $[\![e]\!]$ with Python closures.

3.7.1 Naive

The first execution backend implemented was a *naïve Phi interpreter*. It directly implements the denotational semantics of Phi, so Axial compilation is not necessary. This backend was primarily used as a reference, as it was easy to extend. It focused on correctness rather than performance, and it is order of magnitudes slower than the NumPy backend. For example, staging $\Phi\, i[n].\, 2 \cdot i$ results in a function `(dict[Variable, Value]) -> Value` equivalent to:

```
lambda env: [2 * i for i in range(env[n])]
```

3.7.2 NumPy

We follow the same denotation-guided staging approach as in the naive backend. This time around we apply the whole suite of transformations and use Axial compilation from Phi to Yarr. The key difference is a stricter value representation that must rely on NumPy's `ndarrays`. We set values to be either arrays, or tuples of arrays. On this basis, we also apply some NumPy-specific performance optimisations.

Existing arrays are introduced into Ein programs using the `wrap` function. Alternatively, the programmer can use Ein's `@function` decorator, in which case wrapping and evaluation is performed automatically.

Bindings In some cases, NumPy attempts not to copy arrays, and instead returns a *view*. For instance, `numpy.transpose(arr, (1, 0))` does not copy and instead returns an array with the same underlying data as `arr`, but a transposed layout. This is beneficial when the value is used once, but harmful when it is reused due to cache-inefficiency. We make use of a heuristic approach to address this – whenever a view would be bound to a variable (hence reused), it is immediately copied to the target layout instead.

Ein's let-bindings enjoy more efficient memory management than a manually-written NumPy program. In typical programs, new arrays are allocated sequentially. Afterwards, memory can only be freed by the garbage collector at the end of a function's scope. With explicit let bindings, we achieve higher granularity – arrays can be freed as soon as they are no longer used.

Eliminating temporaries Where they cannot return an array view, by default NumPy operations allocate fresh memory for the result. Operations can be performed in-place instead, but idiomatic NumPy programs are inconvenient to write in this style (and thus waste memory). Consider adding three vectors `(a, b, c)`. The computation `a + b + c` will allocate memory for both `a + b` and `(a + b) + c`. If we can prove `a + b` will not be reused, we can reuse its memory. In NumPy, we would have to write:

```
t = numpy.add(a, b)      # Allocate t = a + b
numpy.add(t, c, out=t)   # Reuse, update t := t + c
```

Ein's NumPy backend performs this in-place update optimisation automatically through a safe heuristic.

3.7.3 Generalising the NumPy backend

The same methods as for NumPy can be used for other libraries which derive from it. **PyTorch** and **JAX** backends were implemented to show this in practice. PyTorch can efficiently execute programs on hardware accelerators (like GPUs), since it implements a suite of well-optimised *kernels*. In the same vein, JAX can perform ahead-of-time compilation through LLVM/XLA for a variety of targets. This fulfils one of the main aims of the project – existing libraries already implement hundreds of thousands of lines of code, and Ein can directly take advantage of this.

Ein backends are selected by calling different variants of the `.eval()` method: `.numpy()`, `.torch()` and `.jax()`. Alternatively, one passes a `backend` keyword argument.

Automatic differentation A feature common to all deep learning frameworks is the capability of performing *automatic differentiation* (AD). Notably, we do not implement it directly in Ein. This choice is motivated by AD's availability in Ein's potential backends. Indeed, to differentiate an Ein function, we could use the PyTorch (or JAX) backend. In the following, we compute $\frac{\partial \mathbf{a}^T \mathbf{b}}{\partial \mathbf{a}} = \mathbf{b}$:

```
import torch
# Initialise a pair of vectors with gradient tracking
a, b = (torch.randn(3).requires_grad_(True) for _ in range(2))
# Compute a dot product in Ein and execute in PyTorch
c = reduce_sum(lambda i: wrap(a)[i] * wrap(b)[i]).torch()
# Run backpropagation through Ein's torch graph
c.backward()
print(a.grad)  # == b
```

Abstract backend To facilitate adding new backends, the `AbstractArrayBackend` interface was implemented. It declares a standard set of array routines, from which a Yarr interpreter may be derived. An excerpt from its definition is given in Figure 3.10.

```python
class AbstractArrayBackend(abc.ABC, Generic[T]):
    ...
    @abc.abstractmethod
    def transpose(self, target: T, permutation: tuple[int, ...]) -> T: ...
    @abc.abstractmethod
    def squeeze(self, target: T, axes: tuple[int, ...]) -> T: ...
    @abc.abstractmethod
    def unsqueeze(self, target: T, axes: tuple[int, ...]) -> T: ...
    @abc.abstractmethod
    def gather(self, target: T, item: T, axis: int) -> T: ...
    ...
```

Figure 3.10: Excerpt from the definition of the array backend interface. For NumPy, `T = numpy.ndarray`.

3.7.4 Extrinsics

There are cases where Ein's compiler does not directly generate calls to functions available in the backend. For instance, `numpy.sort` never gets called directly by Ein. To this end, *extrinsics* were implemented to facilitate calls to backend-specific functions. This is facilitated with the `ext` primitive, which takes a function and its Ein type signature. Extrinsics allow nearly-seamless integration of Ein and library code, vastly improving flexibility. In the below example, we sort an Ein `Vec[Float]` using `numpy.sort`:

```
def sort(x: Vec[Float]) -> Vec[Float]:
    return ext(numpy.sort, ([Vec[Float]], Vec[Float]))(x)
a: Vec[Float] = wrap(numpy.array([2.5, 0.0, 1.0]))
print(sort(a).numpy())  # [0.0, 1.0, 2.5]
```

Ein does have to make some assumptions about the provided function, forming the *extrinsic contract*. Firstly, the function has to be pure – it cannot mutate its arguments. Secondly, it has to be broadcastable over leading axes of the arguments. Lastly, the shape of the array returned by the extrinsic must only depend on the shapes of arguments.

3.8 Repository overview

The Ein repository was built from scratch, and no pre-existing code was used. It follows a usual Python project structure, with the root directory containing the `ein/` source directory, `tests/`, `tools/`, project configuration files (`pyproject.toml`, `.pre-commit-config.yaml`, etc.), and the Git configuration. A high-level breakdown is given in Table 3.1. The repository totals 4815 lines of Python code in the implementation and 2256 in tests and tools (as counted by `cloc`, with the code formatted by `black`). I included an extensive `README.md`, which contains setup instructions and a brief documentation. I also implemented debug tooling for pretty-printing the compiler IR and visualising term graphs (see Appendix C).

Directory	Files	Description	LoC
ein/	symbols.py, value.py, term.py	Abstractions for defining Phi/Yarr	216
ein/phi/	phi.py	Phi grammar and typing rules	648
	type_system.py	Type definitions	245
ein/frontend/	layout.py	Encoding Ein records	167
	ndarray.py, ...	User-facing API, building Phi	782
ein/midend/	substitution.py	Generic substitutions	20
	lining.py	Inlining and outlining	185
	equiv.py, size_classes.py	Size equivalence judgements	195
	structs.py	Array-of-structs to struct-of-arrays transform	172
ein/codegen/	yarr.py	Yarr grammar and typing rules	811
	axial.py	Axials for Yarr	168
	phi_to_yarr.py, ...	Compilation scheme	831
ein/backend/	naive.py	Naïve Phi interpreter	96
	array_backend.py	Abstract array backend definition	255
	numpy_backend.py, ...	Execution backends	670
ein/debug/	graph.py	graphviz diagrams for term graphs	98
	pprint.py	Printing intermediate Phi and Yarr	413
tests/	test_*.py	General and feature-specific tests	1356
tests/suite/	deep/*.py, misc/*.py, parboil/*.py, rodinia/*.py	Longer *cases* for tests and benchmarks	1075
tools/	benchmark.py, ...	Project tools (e.g. benchmarking)	412
.	pyproject.toml, ...	Project configuration files	78

Table 3.1: Overview of the Ein repository.

4 Evaluation

Following the project proposal, evaluation focused on **expressiveness**, **correctness** and **performance**. As part of expressiveness, I consider Ein's usability as an alternative to NumPy, but also its computational limitations. For correctness and performance I rely on diverse test and benchmark suites.

4.1 Expressiveness

We evaluate the expressiveness of Ein from two perspectives:

- For a *programmer*, what patterns are easier to express in Ein than in the established NumPy?
- *Computationally*, what can or cannot be expressed in Phi, Ein's formal program representation?

4.1.1 Programming in Ein

I argue that Ein is a superior alternative to NumPy-like array frameworks in a variety of cases. Ein's design takes advantage of the pointful style and higher-order combinators. It introduces language features unseen in the array programming model. On the other hand, NumPy is occasionally more concise and is much more mature as software, but often leads to bad code for complex operations. To illustrate the key points, we follow a few case studies and discuss the differences between Ein and NumPy.

Logits in a Graph Attention Network

Our flagship example of an unwieldy many-dimensional operation is the logit computation in a Graph Attention Network (GAT), as introduced in the very first chapter. The NumPy implementation below was paraphrased from the original real-world JAX code (the interface of which is based on NumPy):

```
def logits(att_1, att_2, att_e, att_g):
    return (
        transpose(expand_dims(att_1, axis=-1), (0, 2, 1, 3)) +  # + [B, H, N, 1]
        transpose(expand_dims(att_2, axis=-1), (0, 2, 3, 1)) +  # + [B, H, 1, N]
        transpose(att_e, (0, 3, 1, 2)) +                        # + [B, H, N, N]
        expand_dims(expand_dims(att_g, axis=-1), axis=-1)       # + [B, H, 1, 1]
    )                                                           # = [B, H, N, N]
```

We re-express this in the following Ein code:

```
def batch_logits(s, t, e, g):
    return array(lambda h, u, v: s[u, h] + t[v, h] + e[u, v, h] + g[h])

def logits(att_1, att_2, att_e, att_g):
    return array(lambda b: batch_logits(att_1[b], att_2[b], att_e[b], att_g[b]))
```

It is worth remarking that the Ein code compiles into NumPy calls which are just as efficient as the base implementation. We move on to discuss the key differences:

Shape manipulation The array programming model revolves around manipulating array shapes to 'align' them for subsequent whole-array operations. This does not generalise well to **many-dimensional data** – the code becomes polluted with these manipulations (and *axis indexing*), and hence syntax becomes unhelpful in determining the actual semantics. In contrast, Ein code relies on index-oriented descriptions. Extending the implementation with another dimension is just adding another index – rather than modifying fragile arguments to NumPy routines. Additionally, index names serve to hint at the meaning of dimensions – like variable names hint at data they store.

Batching The Ein code is different from what was foreshadowed in the Introduction. We separated out `batch_logits`, which does most of the work, but disregarding the *batch axis* (named `b`). A batch axis is a pervasive code structuring technique in NumPy code, which allows mapping an operation over many inputs (here, comments denote it `B`). It is assumed that it will be carried through in broadcasting. In Ein, we explicitly map elements across each index `b`. This is clearer in intent and avoids arbitrary code conventions.

Min-plus matrix multiplication

We now consider the case of min-plus (or *tropical*) matrix multiplication, which is relevant in shortest path problems. For a pair of matrices A and B, its result is a matrix D given by $D_{i,j} = \min_k A_{i,k} + B_{k,j}$ (like a usual matrix product, but in a $(\min, +)$ semiring instead of $(+, \times)$). We compare equivalently efficient NumPy and Ein implementations that compute this product in $\mathcal{O}(n^2)$ space.

```
def min_plus_product(a, b):
    n,  m = a.shape
    m_, k = b.shape  # m == m_
    d = full((n, k), float("+inf"))
    for t in range(m):
      d = minimum(
        d,
        expand_dims(a[:, t], axis=1) + b[t, :]
      )
    return d
```

```
def min_plus_product(a, b):
    return array(
      lambda i, j: fold(
        float("+inf"),
        lambda k, acc: min(
            acc,
            a[i, k] + b[k, j]
    )))
```

Reasoning about shape In this example, it is necessary to directly access NumPy array shapes to initialise the accumulator and compute the number of iterations. Code clarity requires introducing additional variables `n, m, k` to the code, disconnecting them from `a` and `b` which they describe. In Ein, **size inference** is sufficient to assign the right sizes and iteration counts in common cases.

Data-parallel loops A space-efficient NumPy implementation necessitates a loop with a matrix accumulator. Data-parallel loops like this are simple to express with Ein's `fold`. We avoid point-free reasoning about a matrix accumulator, instead focusing on a fixed index `(i, j)`. Arguably, the `fold` syntax is far from ideal, but readability could be helped with a shorthand:

```
def fold_min(f):
    return fold(float("+inf"), lambda k, acc: min(acc, f(k)))
d = array(lambda i, j: fold_min(lambda k: a[i, k] + b[k, j]))
```

Pairwise L_1 distances

In the last example, we compute the sum of absolute differences (L_1 distance) between every pair of rows of a matrix `A`. We compare NumPy (as in [25]) and Ein implementations:

```
def pairwiseL1(A: ndarray) -> ndarray:
  return sum(
    abs(transpose(A, (1, 0))
        - expand_dims(A, axis=2)),
    axis=1)
```

```
def L1(u: Vec[Float], v: Vec[Float]) -> Float:
  return reduce_sum(lambda i: abs(u[i] - v[i]))

def pairwiseL1(A: Vec[Vec[Float]]):
  return array(lambda i, j: L1(A[i], A[j]))
```

Separation of concerns The Ein implementation separates the computation of L_1 distances into the function `L1` – this is difficult to replicate in NumPy. By forcing reasoning on whole arrays, the array programming model actively discourages encapsulation. For example, say we want to generalise the code to different distances. A higher-order NumPy `pairwiseL1` parameterised by the distance function would necessitate a certain contract (like a batch axis). In contrast, Ein's `pairwiseL1` could be rewritten to take any distance function `dist` with `L1`'s signature:

```
def pairwiseL1(A: Vec[Vec[Float]], dist: (Vec[Float], Vec[Float]) -> Float):
    return array(lambda i, j: dist(A[i], A[j]))
```

Ein's user-defined types also lead to rich abstractions. For instance, it is possible to nearly replicate the automatic differentiation transformation of Shaikhha et al. [33] by implementing an appropriate dual number type. We benchmark the use of abstractions in the 'Semirings' case (Section 4.3).

Type signatures NumPy's support for useful type annotations is poor. Type checking with standard tools like `mypy` is even more problematic, and only partially solved by bespoke extensions. In contrast, Ein's type system is simple, behaves well under composition, and can be checked by `mypy`. For example, `mypy` infers the return type of `pairwiseL1` – `Vec[Vec[Float]]`.

4.1.2 Computation in Phi

Having considered the practicalities of Ein, we consider its computational limitations on the basis of Phi.

Primitive recursion

Phi is not Turing-complete – all programs terminate. A lower bound on its expressive power is given by primitive recursive functions. Indeed, fold directly corresponds to primitive recursion ρ. Nevertheless, we argue Turing-complete computations are seldom necessary in array programming – for instance, no native NumPy routines involve unbounded search. Since Ein is an embedded DSL, we can rely on the generality of its host language, Python.

Lack of in-place updates

A major limitation of Phi is the lack of support for in-place updates (e.g. as in `a[5] = 42`). Since Ein is purely functional, we cannot rely on side effects to handle updates. To this end, Futhark uses a uniqueness type system, while Dex makes use of an effect system [18, 29].

It is worth noting in-place updates can be (unsafely) replicated through the use of extrinsics:

```
def update(vec: Vec[Scalar], p: Scalar, x: Scalar) -> Vec[Scalar]:
    def with_update(arr: ndarray, pos: int, val: int) -> ndarray:
        arr[..., pos] = val; return arr
    return ext(with_update, vec.expr.type)(vec, p, x)
```

Here, `update` mutates its `vec` argument, returning `array(lambda i: where(i != p, vec[i], x))`. Careless use leads to unexpected behaviour, but the approach exemplifies how extrinsics can be used to extend Ein.

Parallelism Ein offers only limited support for common computational patterns. Though associative reductions were implemented as an extension, there are many other parallel programming patterns. These are best exemplified by Futhark's array combinators – particularly scan, histograms (*scatter* in GPGPU programming), and filtering. Only some can be efficiently interpreted by calling out to NumPy routines, pointing to possible extensions to its interface.

4.2 Correctness

Formally proving the correctness of compilers is difficult, and few projects attempt it (e.g. CakeML [22]). I settled for a robust test suite testing runtime correctness.

4.2.1 Tests

Ein's test suite (implemented in the `pytest` framework) mainly consist of compiled program output correctness in a variety of settings. I distinguish three kinds of such tests:

- 13 Phi/IR-level tests. These consist of small program definitions directly in the Phi calculus. They are essentially unit tests for the naïve Phi interpreter.
- 59 Ein tests of small to medium length, which are implemented through Ein's API. This includes feature-specific tests for: extrinsics (4 tests), records (7 tests), and the PyTorch backend (6 tests).
- 9 *case* tests, which are longer. They were constructed from publicly open array programs. We also use these for benchmarking (see Section 4.3).

Correct outputs are generated with Python reference code. To obtain baseline outputs for Phi programs constructed by Ein I use the naïve Phi interpreter. This allows diagnosis whether the problem is with the compiler to Yarr or the source Phi. Outputs are treated as correct even when equality is approximate (by `numpy.testing.assert_allclose`), as many tests use floating point arithmetic. Specifics of floating point are often disregarded in domains like machine learning due to the limitations they cause in optimisations – famously, floating point addition is not associative [4].

Most tests are parameterised by execution backend (through `pytest.mark.parametrise`) and run on random inputs. Since control flow in Phi is limited, random data is sufficient to capture whether compiled programs behave correctly.

The entire test suite passes successfully. The test line coverage across the implementation was measured to be **92%**. Most of the untested code is in error paths, as tests focus on ensuring that correct Ein/Phi programs behave properly. Checking that bad programs fail gracefully would mostly serve to test user experience, which was not part of the success criteria.

4.2.2 Defensive programming

To improve the capability of tests to pick up bad behaviour in internal parts of the implementation, a form of *defensive programming* was applied throughout. This was a general practice of ensuring expected compiler invariants are met through Python `assert` statements. Failing an assertion generally indicates an internal compiler error and hence a bug.

We consider a few examples of this defensive approach:

- Types in the array-of-structs to struct-of-arrays program transformation are mapped in a consistent way ($\mathcal{S}[\![\square(\tau_1 \times \tau_2)]\!] = \square\tau_1 \times \square\tau_2$). We assert this mapping is upheld for each subexpression.
- Outlining asserts the original program p and $\mathrm{inline}(\mathrm{outline}(p))$ are equal. Therefore, we ensure let-bindings inserted preserve program semantics.
- We check that all Phi and Yarr terms constructed are well-typed.

4.3 Performance

Ein's code-generation phase results in a Yarr program, which is by default interpreted by calling respective NumPy routines. Hence, our choice of compilation target (and runtime) is unusual in that it is rather high-level. 'Machine instructions' are NumPy routines, and hence we are limited by NumPy's performance. We thus rely on the established approach of comparing compiler-generated programs against baselines.

I formed a suite of baseline Python programs using NumPy, and compared their performance against equivalent Ein code. The choice of these baselines is inherently subjective – for any Ein-generated program using NumPy, there is a just-as-fast program using NumPy directly. For a fair comparison, I aimed for *idiomatic* NumPy code (e.g. freely allocating temporary arrays instead of managing memory). We consider Ein to be *efficient* if at large problem sizes the running time is within a small constant factor of the baseline.

Suite Our benchmark suite (Table 4.1) builds on the findings from Section 2.6.2. Most cases are sourced from Futhark benchmarks [15] (as done by Dex) and open-source implementations of deep learning architectures. I hand-translated all of the cases from the source language (Futhark, or Python using PyTorch/JAX) into an equivalent pair of Python programs using Ein or NumPy.

A benchmark worth special note is *Semirings*, which was implemented from scratch. The NumPy baseline is a standard $\mathcal{O}(n^3)$ implementation of the Floyd-Warshall all-pairs shortest paths algorithm. In Ein, I instead make rich use of arrays of user-defined dataclass types. I follow the approach of Dolan [10] for solving a general family of problems via matrices over *closed semirings*[1]. I implemented his generic approach in Ein, taking advantage of efficient array operations. NumPy lacks similarly capable abstractions, and in Ein we show they are of zero cost. See Figure 4.1 for the crux of the generic Ein implementation.

[1] A *closed semiring* is an algebraic structure following certain algebraic laws. From an implementation perspective, they are a type which defines some closed binary operations ($+$, $\times$, and closure) and constants (0, 1). They are a useful abstraction – for instance, the *tropical semiring* turns out to capture shortest paths in a graph [24].

Benchmark	Source	Description
Attention	Open-source	core part of Neural Attention
GAT	Open-source	inference in the Graph Attention Network architecture
Semirings	Based on Dolan [10]	shortest paths via closure of a tropical semiring matrix
MRI-Q	Parboil	basic linear reduction
Stencil	Parboil	basic 3D stencil
Hotspot	Rodinia	complex 2D stencil
Pathfinder	Rodinia	1D dynamic programming

Table 4.1: Brief overview of each of the Ein benchmark cases. They are also included in the test suite as system tests. Implementation code of GAT is given in Appendix A as an example.

```
class SemiringMatrix:
    ...
    def closure(self):
        return SemiringMatrix(fold(
            self.elem,
            lambda k, acc: array2(
                lambda i, j: acc[i, j] + acc[i, k] * acc[k, k].closure() * acc[k, j]
        ).assume(self.elem[k, k]))) + self.one
```

Figure 4.1: Closure of a `SemiringMatrix` in Ein (Algorithm 6.1 [2]). $+$, $\star$ are overloaded `Semiring` operations. When $+$ is min and $\star$ is $+$, this reduces to the Floyd-Warshall shortest paths algorithm.

It is worth note that the other two cases used in the test suite – NN and KMeans (from Rodinia) – were excluded from the benchmark. A computationally efficient implementation of these relies on features unavailable in Ein (respectively in-place updates and histogram computations). They could be mimicked through extrinsics in practice, but this would not constitute an informative benchmark.

Methodology Benchmarks were conducted on an M1 Pro MacBook (14-inch, 2021). The versions used were CPython `3.11.9` and NumPy `1.26.4`. No thread parallelism was used beyond what is used by default in NumPy (in linear algebra routines). Large problem sizes were used (about 1 s runtime), taking the minimum time across at least 5 runs. Confidence intervals are given by the geometric standard deviation taken across runtime ratios of independent runs of Ein and the baseline. The ratio of minimum runtimes is reported, as each benchmark performs a deterministic sequence of operations, so most overheads are caused by external factors. Compilation was not included in the measurement, but it is negligible (~ 2 ms).

Discussion Benchmark results can be seen in Figure 4.2. We reach the performance goal – Ein is no more than 60% slower than the baseline, and it can be up to 50% faster. Ein's is able to perform various optimisations – eliminating allocations of temporary arrays is particularly important. On the other hand, performance deficits are especially due to missing advanced indexing strategies – especially in Stencil. However, the baseline Stencil implementation is ridden with error-prone expressions. It contains nearly a dozen complex indexing expressions like `A[1:-1, :-2, 1:-1]`, which in Ein would translate to `A[x, y - 1, z]`. Therefore, we conclude Ein offers a good trade-off on code readability and performance.

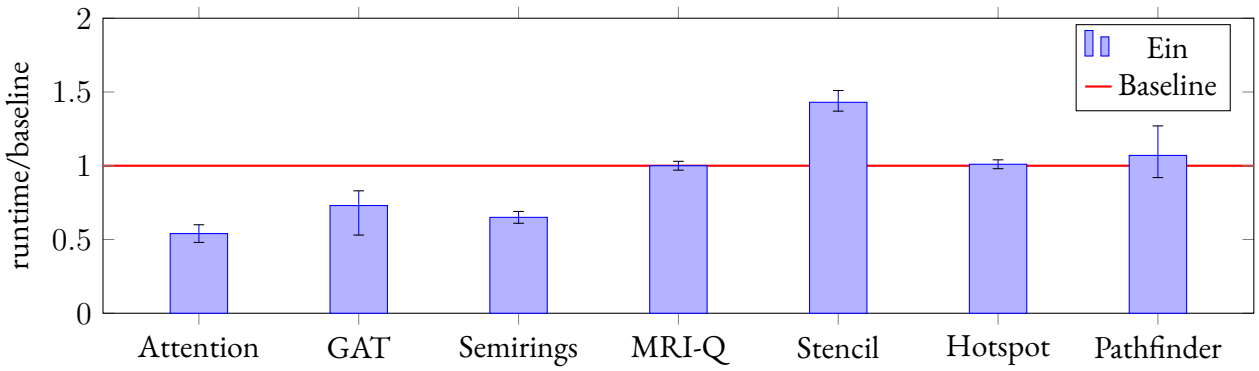


Figure 4.2: Ratio of the running time of Ein-generated NumPy programs, and the baseline NumPy. *Lower is better.* Expanded results for varying problem sizes are given in Appendix B.

4.4 Work completed

We summarise the work completed basing on the success criteria from the proposal:

Designing a pointful array calculus Phi was designed successfully, and ended up building on the $\tilde{F}$ language due to Shaikhha et al. [33]. This further includes various extensions with respect to the proposal: fully-fledged pair types, size assertions, folds, and associative reductions.

Embedding an array language in Python The `ein` library successfully embeds a pointful array language – Ein – in Python. Ein's API builds up terms in the Phi calculus, after which it allows evaluating the program with an execution backend. Key extensions are: size inference, general `fold` and `reduce` combinators, extrinsics, rich record types, and support for type annotations.

Efficient execution backend We compile pointful array programs (in the Phi calculus) by generating point-free array code (Yarr calculus) using the foundation of the newly introduced Axials. Yarr programs are efficiently interpreted by calling NumPy routines. A key extension was the inclusion of a additional backends – using PyTorch and JAX rather than NumPy – allowing Ein to take advantage of hardware acceleration and automatic differentation.

Ein's compiler middle-end was also suitable for extensions: the array-of-structs to struct-of-arrays transformation and outlining (common subexpression elimination and loop-invariant code motion).

4.5 Summary

Throughout this chapter we have seen that Ein is expressive, and can be superior in usability as an alternative to NumPy. It further has various language features unseen in NumPy-like frameworks. Furthermore, robust tests ensure correctness of our implementation. Benchmarking has shown our execution backend is efficient on a varied suite of cases. Ein leads to much clearer programs than what would be written by a NumPy programmer, while preserving – or even improving – performance.

5 Conclusions

The project was a complete success. I have met all of the success criteria, and greatly expanded upon them (Section 4.4). I introduced novel techniques that allow the reconciliation the engineering efforts on the array programming model and the promises of pointful array programming. Through them, I embedded language features in Python one might not expect to be possible with existing array libraries. Due to Python's dominant role in machine learning and scientific computing, these techniques can have real-world impact. On this basis, Ein represents an advance of array programming in Python.

5.1 Publications

I submitted papers on the topic of this project to two venues – the ACM Student Research Competition at POPL (November – January), and the ARRAY workshop at PLDI (March – April).

5.1.1 Student Research Competition

The ACM Student Research Competition (SRC) consists of conference SRCs and the Grand Finals for winners at each conference. In November, I submitted a 3-page extended abstract to the SRC at the 51st ACM SIGPLAN Symposium on Principles of Programming Languages (POPL), and received an invitation to present my work in January at the conference. There, I gave poster and oral presentations. My supervisor supported me throughout. I was awarded **first prize** among undergraduates, and hence invited to submit a 5-page abstract to the Finals, with results due later in May.

5.1.2 ARRAY Workshop

The ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming (ARRAY) concerns all aspects of array programming, and is colocated with the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). Together with my supervisor we authored a 12-page research paper for the 10th ARRAY workshop in early April. I wrote the main content, with my supervisor taking an advisory role. The submission was **accepted** for publication.

5.2 Lessons learnt

Ein was by far my longest-running software project, and the first one which involved such a significant theoretical and technical background. It was necessary to study the relevant concepts in programming languages and compilers. Having worked with large codebases prior, the software engineering aspect did not pose as much of a challenge. On the other hand, designing Ein – and the foundations underlying it, Phi and Yarr – was far from straightforward. Describing my design well also proved challenging.

5.2.1 Designing programming languages

Ein sought to be a domain specific language for array programming – targeting practical use in areas like machine learning – embedded in Python. Each of these constraints posed problems.

Domain-specific language Learning to choose the right **trade-offs** for my DSL was a major difficulty. Whenever I thought about a new feature, I was constrained by its impact on the rest of the language. I had to make careful choices – for instance, including the simpler `fold` rather than a general while-loop led to a compilation scheme preserving data parallelism.

Array programming It was important to relate my project goals with what *ought to be* possible in the language based on its domain. To this end, I sought feedback from my colleagues in machine learning, and consulted external resources. This taught me reconciling real-world use and language design.

Embedding in Python Python was not explicitly designed to host embedded DSLs. Though it is extremely flexible (it is dynamically typed, interpreted and allows introspection), it is constrained in two main areas – function overloading and variable introduction. Finding a way to include new syntactic features that would be idiomatic *(Pythonic)* was a source of trade-offs.

5.2.2 Expressing ideas

Throughout this project I explored many different ideas for language features. Unfortunately, I often lacked the necessary background to give them solid theoretical foundation. While implementation was a matter of software engineering, formalising ideas and communicating them across was challenging.

Throughout the project, under encouragement of my supervisor, I wrote and submitted two papers (Section 5.1). This forced me to reflect on my ideas and find ways of explaining them succinctly.

5.3 Future work

Avenues for further work include:

Generalising the compilation scheme Can the Axial compilation scheme be further generalised to other combinators than array comprehensions, folds and associative reductions? Is it possible to integrate this with irregular parallelism or jagged arrays?

Alternative backends Is it beneficial to compile Phi to a different representation than Yarr? Can we take advantage of existing work for Dex and $\tilde{F}$ by implementing different backends [29, 33]?

In-place updates What is the best way of implementing safe in-place updates in the context of an embedded DSL? JAX, Futhark and Dex offer possible solutions [11, 18, 29].

I plan to release the source code of the `ein` library under an open licence, so that it may be used in practice and its methods adapted into new tooling.

Bibliography

1. M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. "TensorFlow: a system for large-scale machine learning". In: *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 2016, pp. 265–283.
2. S. K. Abdali and B. D. Saunders. "Transitive closure and related semiring properties via eliminants". *Theoretical Computer Science* 40, 1985, pp. 257–274.
3. K. Åhlander. "Einstein summation for multidimensional arrays". *Computers & Mathematics with Applications* 44:8-9, 2002, pp. 1007–1017.
4. M. B. Alawi. "What Every Programmer Should Know about Floating Point Arithmetic". PhD thesis. University of Leeds, School of Computing, 2004.
5. R. Atkey, S. Lindley, and J. Yallop. "Unembedding domain-specific languages". In: *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*. 2009, pp. 37–48.
6. R. Bernecky. "Fortran 90 Arrays". *ACM SIGPLAN Notices* 26:2, 1991, pp. 83–98.
7. D. Chiang, A. M. Rush, and B. Barak. "Named Tensor Notation". *Transactions on Machine Learning Research*, 2022.
8. M. Cole. *Parallel programming, list homomorphisms and the maximum segment sum problem*. Citeseer, 1993.
9. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2022.
10. S. Dolan. "Fun with semirings: a functional pearl on the abuse of linear algebra". In: *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*. 2013, pp. 101–110.
11. R. Frostig, M. J. Johnson, and C. Leary. "Compiling machine learning programs via high-level tracing". *Systems for Machine Learning* 4:9, 2018.
12. J. Gibbons. "Aplicative programming with naperian functors". In: *Proceedings of the 1st International Workshop on Type-Driven Development*. 2016, pp. 13–14.
13. J. Gibbons and N. Wu. "Folding domain-specific languages: deep and shallow embeddings (functional pearl)". In: *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*. 2014, pp. 339–347.
14. G. Guennebaud, B. Jacob, et al. *Eigen v3*. http://eigen.tuxfamily.org. 2010.
15. T. F. Hackers. *Futhark Benchmarks*. URL: https://github.com/diku-dk/futhark-benchmarks.
16. C. R. Harris, K. J. Millman, S. J. Van Der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, et al. "Array programming with NumPy". *Nature* 585:7825, 2020, pp. 357–362.
17. T. Henriksen and M. Elsman. "Towards size-dependent types for array programming". In: *Proceedings of the 7th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*. 2021, pp. 1–14.

18. T. Henriksen, N. G. Serup, M. Elsman, F. Henglein, and C. E. Oancea. "Futhark: purely functional GPU-programming with nested parallelism and in-place array updates". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2017, pp. 556–571.
19. P. Hudak. "Building domain-specific embedded languages". *ACM Computing Surveys* 28:4es, 1996, 196–es.
20. K. E. Iverson. "A programming language". In: *Proceedings of the May 1-3, 1962, spring joint computer conference*. 1962, pp. 345–351.
21. K. E. Iverson. "Notation as a tool of thought". In: *ACM Turing award lectures*. 2007, p. 1979.
22. R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. "CakeML: a verified implementation of ML". *ACM SIGPLAN Notices* 49:1, 2014, pp. 179–191.
23. Ł. Lachowski et al. "On the complexity of the standard translation of lambda calculus into combinatory logic". *Reports on Mathematical Logic* 53, 2018, pp. 19–42.
24. D. J. Lehmann. "Algebraic structures for transitive closure". *Theoretical Computer Science* 4:1, 1977, pp. 59–76.
25. D. Maclaurin, A. Radul, M. J. Johnson, and D. Vytiniotis. "Dex: array programming with typed indices". In: *Program Transformations for ML Workshop at NeurIPS 2019*. 2019.
26. C. McBride and R. Paterson. "Applicative programming with effects". *Journal of functional programming* 18:1, 2008, pp. 1–13.
27. A. Meurer, A. Reines, R. Gommers, Y.-L. L. Fang, J. Kirkham, M. Barber, S. Hoyer, A. Müller, S. Zha, S. Shanabrook, et al. "Python Array API Standard: Toward Array Interoperability in the Scientific Python Ecosystem", 2023.
28. A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. "PyTorch: An imperative style, high-performance deep learning library". *Advances in neural information processing systems* 32, 2019.
29. A. Paszke, D. D. Johnson, D. Duvenaud, D. Vytiniotis, A. Radul, M. J. Johnson, J. Ragan-Kelley, and D. Maclaurin. "Getting to the Point: index sets and parallelism-preserving autodiff for pointful array programming". *Proceedings of the ACM on Programming Languages* 5:ICFP, 2021, pp. 1–29.
30. G. Ramalingam and T. Reps. "An incremental algorithm for maintaining the dominator tree of a reducible flowgraph". In: *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1994, pp. 287–296.
31. A. Rogozhnikov. "Einops: Clear and reliable tensor manipulations with einstein-like notation". In: *International Conference on Learning Representations*. 2021.
32. S.-B. Scholz. "Single Assignment C – Functional programming using imperative style". In: *Proceedings of IFL*. Vol. 94. 1994.
33. A. Shaikhha, A. Fitzgibbon, D. Vytiniotis, and S. Peyton Jones. "Efficient differentiable programming in a functional array-processing language". *Proceedings of the ACM on Programming Languages* 3:ICFP, 2019, pp. 1–30.
34. K. Smillie. "A lecture on Array languages". *ACM SIGAPL APL Quote Quad* 30:3, 2000, pp. 14–24.
35. N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen. "Tensor Comprehensions: Framework-agnostic high-performance machine learning abstractions". *arXiv preprint arXiv:1802.04730*, 2018.

A Example benchmark code

In this appendix, we consider the Python implementation of the GAT benchmark case – both using Ein and NumPy. The case is based on an open-source implementation (in `deepmind/clrs` on GitHub) of Graph Attention Networks which used JAX, and which I translated by hand.

The excerpts from the implementations omit the surrounding test harness – all free variables should be assumed to be function arguments, and imports (such as `from ein import array`) are presumed.

Ein

```
def softmax(x: Vec[T]) -> Vec[T]:
    x_max = fold_max(lambda i: x[i])
    x1 = array(lambda i: (x[i] - x_max).exp())
    x1_sum = fold_sum(lambda i: x1[i])
    return array(lambda i: x1[i] / x1_sum)
def leaky_relu(x: Scalar) -> Scalar:
    return where(x < 0.0, 0.01 * x, x)
bias = array(lambda b, u, v: (adj[b, u, v] - 1.0) * 1e9)
logits = array(lambda b, h, u, v: s[b, u, h] + t[b, v, h] + e[b, u, v, h] + g[b, h])
coefs = array(
    lambda b, h, u: softmax(array(lambda v: leaky_relu(logits[b, h, u, v]) + bias[b, u, v]))
)
return array(lambda b, u, h, f: fold_sum(lambda v: coefs[b, h, u, v] * vals[b, v, h, f]))
```

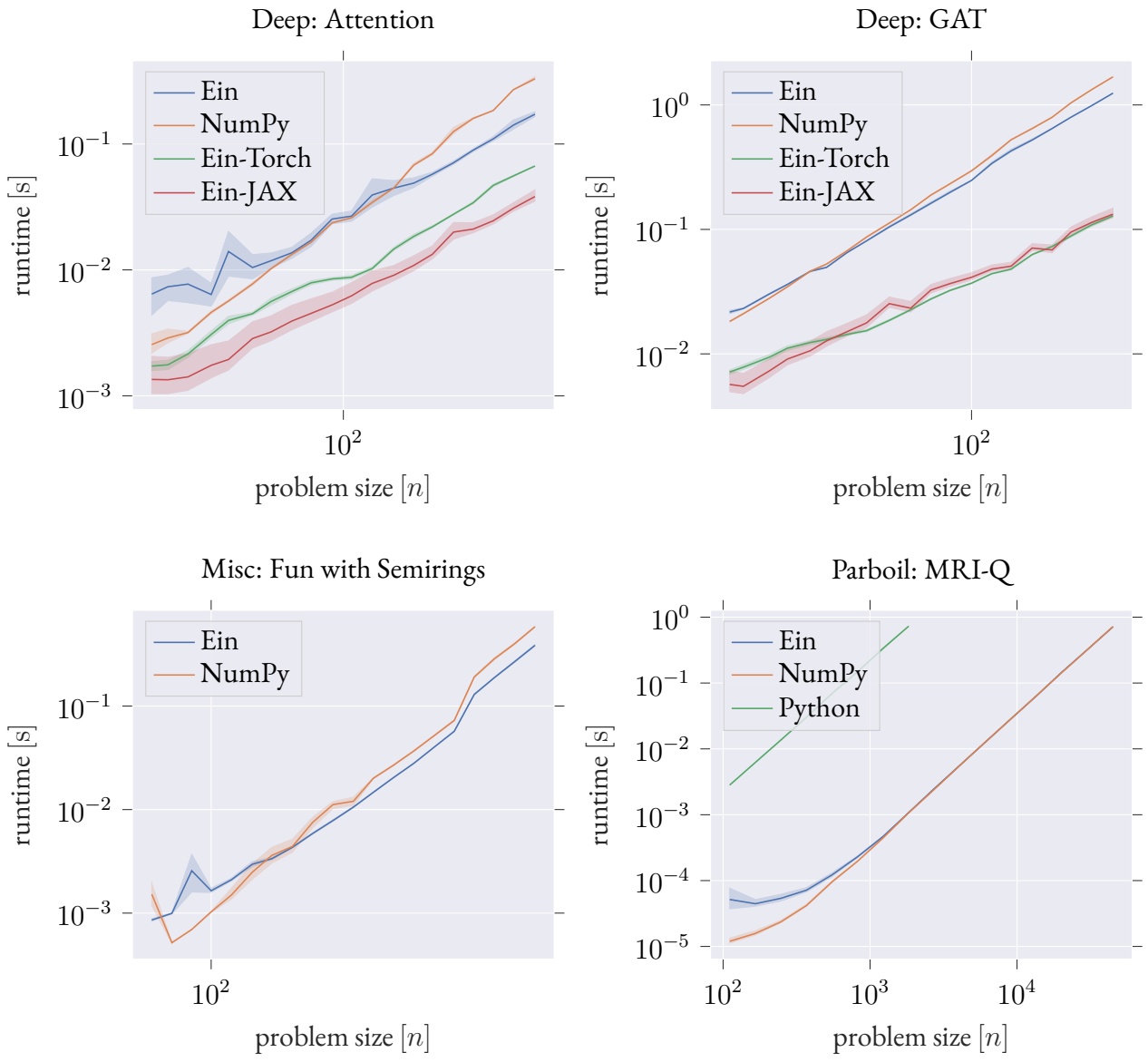
NumPy

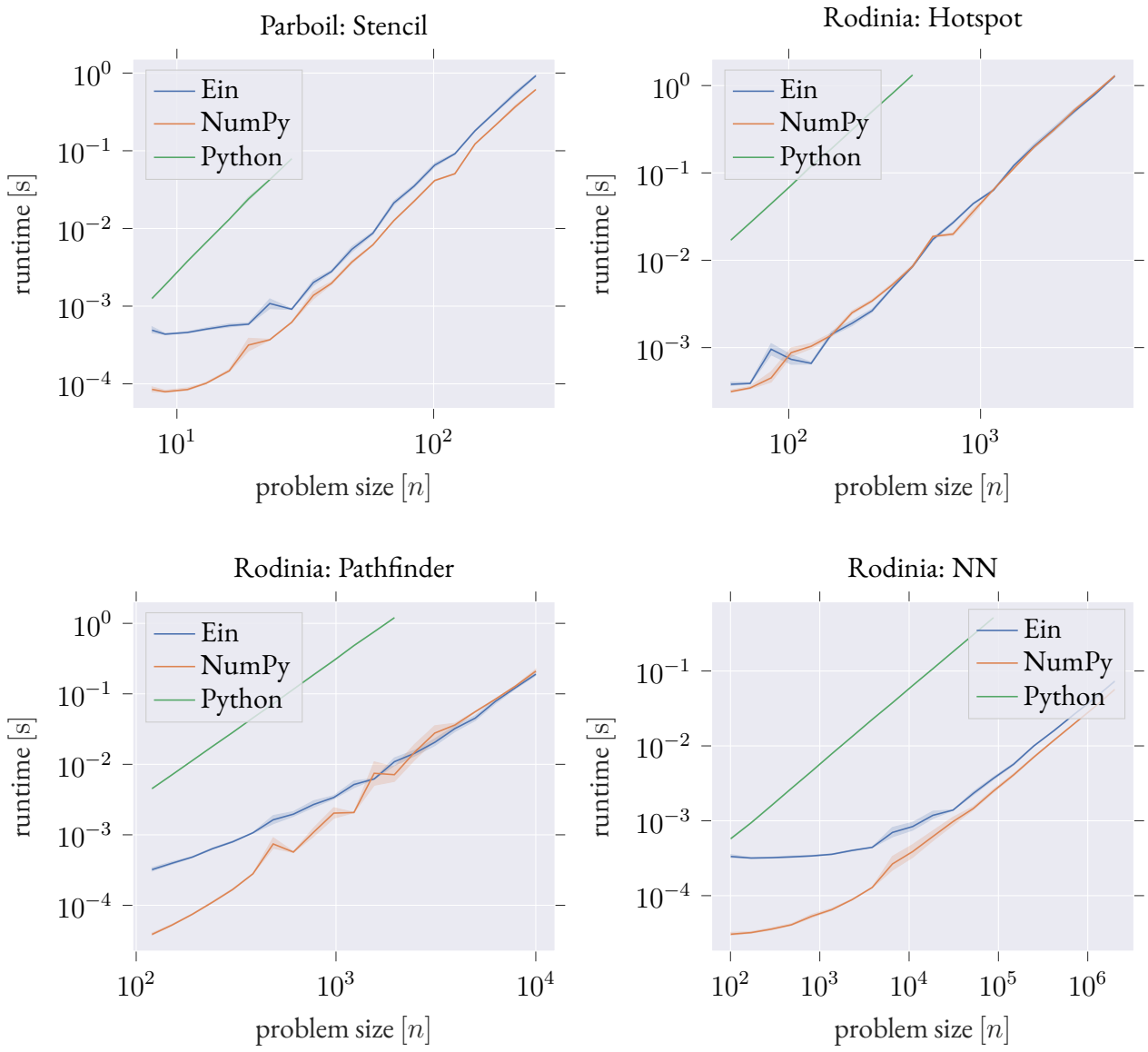
```
batches, vertices, heads = s.shape
bias = (adj - 1.0) * 1e9
bias = numpy.tile(bias[..., None], (1, 1, 1, heads))  # [B, N, N, H]
bias = numpy.transpose(bias, (0, 3, 1, 2))  # [B, H, N, N]
vals = numpy.transpose(vals, (0, 2, 1, 3))  # [B, H, N, F]
logits = (
    numpy.transpose(s[..., numpy.newaxis], (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]
coefs = scipy.special.softmax(leaky_relu(logits) + bias, axis=-1)
ret = numpy.matmul(coefs, vals)  # [B, H, N, F]
return numpy.transpose(ret, (0, 2, 1, 3))  # [B, N, H, F]
```

B Benchmark results

This appendix provides extended benchmark results generated across many problem sizes, depicting any evident uncertainties and overheads associated with Ein. This solidifies arguments that Ein's performance is consistently at an satisfactory (though not perfectly optimal) level.

This data further includes performance measurements for Ein's alternative backends – PyTorch and JAX – on cases based on deep learning domain programs. We did not consider this when evaluating Ein itself, as these backends have characteristics entirely independent from NumPy, hence comparisons to NumPy baselines would be unfair. Since JAX applies ahead-of-time compilation, it performs best of all.




Parboil: Stencil
Rodinia: Hotspot
Rodinia: Pathfinder
Rodinia: NN
Ein
NumPy
Python
runtime [s]
problem size [n]

C Compiler outputs

This brief appendix concerns the compiler outputs for a short example program. Intermediate representations in Phi and Yarr are printed using the compiler's debug utilities. For clarity, a hand-written Python program using NumPy which is equivalent to the Yarr code is given.

Ein

We use Ein's Python API to implement `pairwiseL1`. Type annotations are applied, and the code is successfully type-checked by `mypy`.

```
from ein import array, fold, Vec, Int, Float
from typing import Callable

def fold_sum(f: Callable[[Int], Float]) -> Float:
    return fold(wrap(0.0), lambda i, acc: acc + f(i))

def L1(u: Vec[Float], v: Vec[Float]) -> Float:
    return fold_sum(lambda i: abs(u[i] - v[i]))

def pairwiseL1(A: Vec[Vec[Float]]) -> Vec[Vec[Float]]:
    return array(lambda i, j: L1(A[i], A[j]))
```

Phi

We construct the Phi term for `pairwiseL1` via `ein.function(pairwiseL1).phi(ein.ndarray_type(2, float))` The term is printed as follows (`&x` denote variables, where `&0` is the argument `A`, `@i` are indices):

```
let &1: int = size[0](&0) in
let &2: int = size[1](&0) in
for @0[&1].
  for @1[&1].
    fold &3[&2] init &4: float = 0.0 by &4 =>
      &4 + (let &5: float = &0[@0][&3] - &0[@1][&3] in
            if less(0.0, &5) then &5 else -&5)
```

To make the example more interesting, the absolute value is computed here via a conditional:

$$|x| = \text{where}(0 < x, x, -x)$$

Yarr

We apply code generation on Phi to obtain Yarr code, which uses primitives similar to NumPy routines.

```
let &6: int = size[1](&0) in
let &7: [][]float = unsqueeze((0, 1), 0.0) in
let &8: [][]float =
  let &9: int = size[0](&0) in
  repeat(1, &9, repeat(0, &9, &7))
in
fold &3[&6] init &4: [][]float = &8 by &4 =>
  add(&4,
    let &10: [][]float =
      let &11: []float = take(&0, None, &3) in
      subtract(unsqueeze((1,), &11), &11)
    in
    where(less(0.0, &10), &10, negative(&10))
  )
```

NumPy

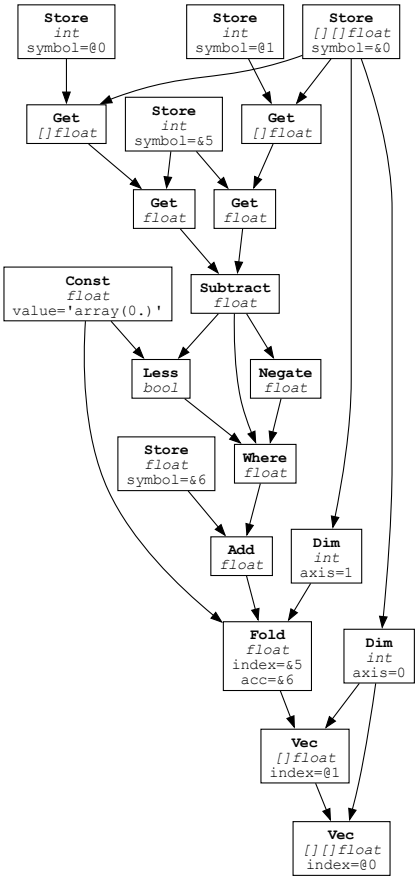
I hand-simplified the Yarr output into an equivalent Python program using NumPy directly.

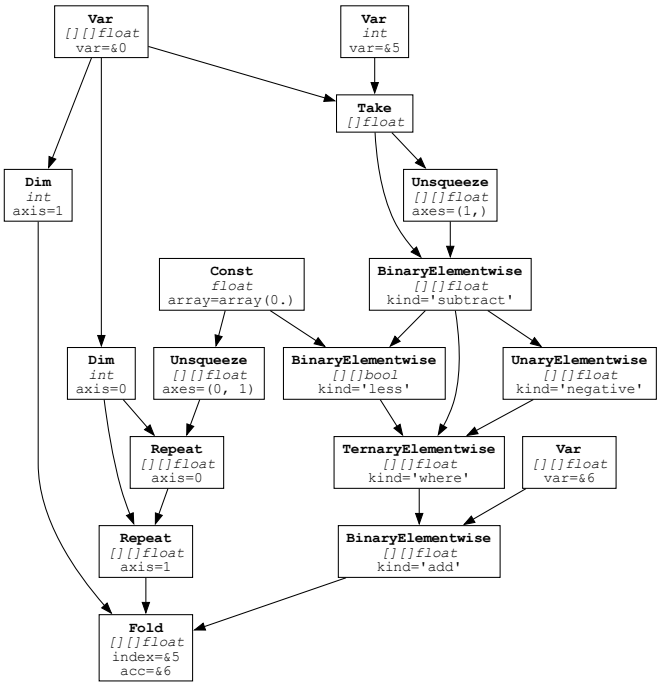
```
from numpy import *

def pairwiseL1(x0: ndarray) -> ndarray:
    x6, x9 = x0.shape[1]
    x4 = zeros((x6, x9))
    for i in range(x1):
        x11 = x0[:, i]
        x10 = expand_dims(x11, axis=1) - x11
        x4 += where(x10 < 0, x10, -x10)
    return x4
```

Term graphs

The Phi and Yarr term graphs, rendered using `graphviz` on graphs generated by `pydot`:





D Project Proposal

The following is quoted verbatim. However, formatting and citations were adjusted to be consistent with the rest of this document.

D.1 Introduction

Tensors are an ubiquitous abstraction in the area of machine learning. In this context, they are n-dimensional arrays specialised for high-performance data parallel computing. The Python programming language dominates the machine learning research scene. However, its runtime overheads are far too high for the data processing to be expressed directly in the language. As such, Python highly benefits from the tensor abstraction with a set of highly optimised operations. This gave rise libraries and frameworks such as NumPy [16], PyTorch and TensorFlow.

These aforementioned libraries all share a common core of the *array programming model*, as introduced in APL [20]. The model is characterised by first-class multi-dimensional arrays. Primitives are parameterised in nontrivial ways, acting on the arrays as a whole – individual elements are rarely referenced. The core benefit of this approach is that the implementations of individual primitives – referred to as *kernels* – can be highly optimised (and written in another language). Programs in the model can also be surprisingly concise, as can be seen on Figure D.1.

```
def pairwiseL1(a):
    return sum(abs(a.T - a[..., newaxis]), axis=1)
```

Figure D.1: Computing a matrix of pairwise L_1 distances between rows with axis manipulation and broadcasting in NumPy – as given by Dex authors [25]

Despite its widespread use in today's scientific computing and machine learning workflows, the array programming model is not without its problems. The available operations expose features such as broadcasting, which allow high expressiveness without sacrificing performance. However, these features also make programs difficult to write, debug and maintain [29]. They are also nontrivial to statically analyse due to being highly dynamic, as the behaviour of operators may vary based on the shapes and element types of their inputs . This causes the need for comments in the host language, as few facilities exist for type annotations . Additionally, the actual implementation in terms of the available kernels may be distant from the mathematical description of a computation – as can be seen in Figure D.2. The choice of kernels is not obvious, as their number is usually in the hundreds. More involved pieces of logic have to be expressed with more specialised kernels (if they exist), or otherwise complex combinations of existing ones (suboptimal performance), or worst of all expressed directly in Python, which significantly hurts performance.

All in all, these shortcomings cause novel deep learning architectures to be unreasonably difficult to express and optimise in the array programming model. This invites the exploration of new array programming paradigms which seek to solve these problems.

Dex [29] is a recent research programming language implementing *pointful* (or *index-oriented*) array programming, which can be seen as a generalisation of *Einstein summation notation*. Its distinctive feature

$$\texttt{pairwiseL1}(A)_{i,j} = \sum_k |A_{i,k} - A_{j,k}|$$

Figure D.2: An index-oriented mathematical formulation of `pairwiseL1` from Figure D.1

is the `for` primitive, where an array's elements may be defined in terms of their indices. Programs written with this approach boast closeness to mathematical notation (Figure D.3). Index bounds are inferred as in Einstein summation for clarity. Dependent types fulfil a central role for statically analysing array sizes. Dex demonstrates that the paradigm can achieve performance comparable to other approaches, such as the *array-combinator* language Futhark [18].

```
def pairwiseL1 (a : m=>n=>Float) : n=>n=>Float =
    for i. for j. sum for k. abs (a.i.k - a.j.k)
```

Figure D.3: Computation of `pairwiseL1` from Figure D.1 in Dex

However, application of an entirely new language like Dex poses further problems, such as its usability in the existing ecosystem. Embedding the DSL [13] is a popular approach to address this. Jax [11] can be seen as a domain-specific language within a shallow embedding in Python. Operations performed on Jax arrays are *traced* into a staged expression. That IR can then be e.g. differentiated, and compiled for execution via XLA . This approach is in contrast to the deep embeddings used in TensorFlow and PyTorch, which manipulate the Python AST. Shallow embeddings are generally better composable and more predictable.

D.2 Description

The main goal of the project is to design and implement a pointful array programming domain-specific language in a shallow Python embedding, lending itself to being easy to use, statically analyse, and execute efficiently.

The basis of the DSL would be formed by an underlying calculus – provisionally named *Ein* and *Phi* respectively. Phi could be based on the semantics of Dex. A proposed basic version of this calculus is shown in Figure D.4. Its core feature is vector comprehensions – defining tensors in terms of their elements as functions of indices. It is not an aim for the core language to include all Dex features such as algebraic effects and dependent types, though similar ideas can be developed as extensions. Phi can be extended with more arithmetic functions and reductions (like product).

A computation in Ein should be traced, and then staged for execution by a *backend*. As NumPy is universally available and has good performance on CPU, it is planned to be the primary backend. Hence, a core part of the project is compilation of expressions in the DSL to an efficient sequence of NumPy calls necessary to execute it.

Though Python is the host language, compilation can happen in another language – for instance Rust, as it interfaces well with Python via PyO3 and is better suited for compiler-related tasks with static types and ADTs. I plan to resolve this in early stages of the project.

$$
\begin{array}{rll}
e ::= & \Phi\, i[e].\, e & \text{(vector comprehension)} \\
\mid & \Sigma\, i[e].\, e & \text{(vector summation)} \\
\mid & e[e] & \text{(indexing)} \\
\mid & e + e \quad \mid \quad e \cdot e \quad \mid \quad e - e \quad \mid \quad e \,/\, e & \text{(scalar arithmetic)} \\
\mid & i \quad \mid \quad x & \text{(index value, variable)}
\end{array}
$$

$$\texttt{pairwiseL1}(a) = \Phi\, i[n].\, \Phi\, j[n].\, \Sigma\, k[m].\, \texttt{abs}(a[i][k] - a[j][k])$$

Figure D.4: Proposed structure of the Phi calculus. Below an example term is given, where a has shape $n \times m$. All three variables are given in scope.

```
n, m = a.shape
# tensor and sum are primitives in Ein, corresponding to phi and sigma in Phi
dist = tensor[n, n](lambda i, j: sum[m](lambda k: abs(a[i, k] - a[j, k])))
# dist may be computed with NumPy arrays:
x = a.reshape(n, 1, m)  # x[i, *j, k] = a[i, k]
y = a.reshape(1, n, m)  # y[*i, j, k] = a[j, k]
z = x - y               # z[i, j, k] = a[i, k] - a[j, k]
w = z.abs()             # w[i, j, k] = abs(a[i, k] - a[j, k])
dist = w.sum(axis=2)    # dist[i, j] = sum(k: abs(a[i, k] - a[j, k]))
```

Figure D.5: Code example in Ein, along with a sequence of NumPy calls for executing it.

D.3 Starting point

I have no experience designing or implementing a machine learning domain-specific language or compiler. I expect the knowledge from Part IB Semantics, Compiler Construction and Concepts courses will be useful during design and development, and that the Part II Denotational Semantics and Types may also be helpful when working on extensions to the semantics and type system of the language. Part IA Scientific Computing and Part IB Data Science gave me an insight into array programming in NumPy itself, though I have also used NumPy in personal projects.

I have been exposed to the domain of machine learning during my open-source work on the ONNX project, which is an industry standard IR for describing models. Working on a research project on Graph Neural Networks during Part IB has also given me some domain experience. I have years of Python experience, including professional, and basic experience with Rust.

Prior to starting the project I had done a fair amount of research into the topic of array programming, including papers on index-oriented techniques and compiler optimisations relevant to linear algebra and machine learning. I had not written any project-specific code before 1 October, though I have written experiments to test the feasibility of some of the ideas.

D.4 Goals

D.4.1 Core

Formalisation Design and formalise operational semantics of a calculus for expressing pointful array programming. A basic type system (up to tensor element type and rank) to validate these terms should be given. These may be based on a subset of the rules presented in Paszke et al. [29].

Embedding Implement a Python API for constructing Phi calculus terms, forming Ein. The terms should be easy to serialise, to make way for tooling developed in other languages.

Execution Create a NumPy execution backend for evaluating Phi expressions. The computation should be orchestrated into a sequence of necessary calls.

Testing Collate an Ein test and benchmark suite, based on the ones in Dex and Futhark. Results should be compared to NumPy implementations in terms of correctness and performance.

D.4.2 Extensions

Optimisation The terms in the calculus benefit from relative expressiveness, and their purity makes them easy to manipulate. Many optimisations could be derived with equality saturation , for instance as implemented in the egg library . A focus would be on optimisations which are not performed by existing frameworks, for instance in linear algebra . This may also include loop fusion, finding common subexpressions (up to arrays), and axis ordering.

Backends Execution with different compatible frameworks in the style of Einops [31] is a large benefit which allows using the DSL within an existing project – for instance one employing PyTorch or Jax. These are relatively similar to NumPy, and so the implementation should be as well. In the direction of a classical compiler, implementing C code generation or compiling directly with LLVM is also possible for portability and slightly better performance. Lastly, state-of-the-art deep learning compilers like XLA or TVM could be used directly.

Types Dynamic sizes in array types are difficult to design types for. Possible solutions include a unification-based approach with existential types , or Futhark's size-dependent types [17]. Advanced static analyses could be done with an SMT solver like Z3 . Another direction is basic parametric polymorphism for numeric types in the style of Haskell's `Num` and `Floating` typeclasses.

Further expressiveness Capabilities like domain-specific intrinsics (e.g. matrix inverse), allowing usage of sequential loops or jagged arrays, and custom reductions (besides summation).

Program transformation Ranges from basic syntactic sugar for inferring index bounds deriving from Einstein summation, up to automatic differentiation (forward or backward) which may base on the approach in Dex [29] or Jax [11].

D.5 Success criteria

- Presenting a design of the Phi pointful array calculus. An integral part is definition of tensors with their elements defined in terms of their indices, as in Dex [29].
- A shallow embedding of the Ein DSL in Python, with an API that constructs Phi calculus terms via tracing. The technique may draw on Jax [11].
- Implementation of an execution backend for Ein, which compiles the terms into a form that may leverage libraries in the Python ecosystem.
- An evaluation of correctness, performance and expressiveness of Ein.

D.6 Evaluation strategy

The evaluation would be performed on a test and benchmark suite derived from the ones in existing projects – such as Dex [29] and Futhark [18]. They themselves are based on other existing suites like Parboil . Other sources of program samples include modern deep learning model architectures, like transformers or graph neural networks.

Correctness The test suite can be used to demonstrate general correctness of the language. If an execution backend is not trivially derived from the operational semantics, a formal proof of correctness may also be given.

Performance Performance should be comparable to existing approaches when the code is casually written without additional care given to performance. Reaching state-of-the-art performance is not within the scope of the project.

Expressiveness Successful creation of a representative test suite will demonstrate the expressiveness of the language. It should be noted that instances of sequential logic (where usage of control flow primitives like algebraic effects in Dex or `loop` in Futhark is necessary) are considered an extension. This is motivated by their relative rarity in deep learning frameworks, where they are seldom the focus of optimisation.

D.7 Timetable

D.7.1 Michaelmas Term

- **Weeks 2 – 4** (October 12th – November 1st)
  - Review literature on tensor languages and index-oriented approaches, particularly Dex.
  - Construct the semantics and type system of Phi.
  - Ensure the type system meets required properties, formulate a Progress theorem.
  - **Milestone.** Formal pointful array programming calculus as a basis for Ein.
- **Weeks 5 – 6** (November 2nd – November 15th)
  - Implement an API for constructing Phi calculus terms.

  - Create a fundamental backend for evaluating terms in pure Python without the usage of an array library.
  - Construct small correctness test cases suitable for the basic backend.
  - **Milestone.** Ein embedded in Python with basic evaluation.

- **Weeks 7 – 8** (November 16th – November 29th)
  - Design the NumPy execution backend and construct a term evaluation algorithm.
  - Prove the correctness of term evaluation under reasonable assumptions.
  - Implement the NumPy staging and execution backend.
  - **Milestone.** NumPy execution backend passing basic tests.

D.7.2 Michaelmas Vacation

- **Weeks 1 – 3** (November 30th – December 20th)
  - Collate a test and benchmark suite from the Futhark, Dex and other codebases.
  - Based on testing feedback, fix and improve on the NumPy backend.
  - *Extensions.* Implement syntactic sugar (e.g. bounds inference) to ease writing the suite.
  - **Milestone.** NumPy execution backend tested and effective.
- **Weeks 4 – 5** (December 21st – January 3rd) – Break for Christmas.
- **Weeks 6 – 7** (January 4th – January 17th)
  - Test, refactor and document the existing code.
  - Finish and improve any core steps.
  - Begin work on progress report.
  - **Milestone.** *Success criteria met.*

D.7.3 Lent Term

- **Weeks 1 – 2** (January 18th – January 31st)
  - Write the progress report and start work on slides.
  - *Extensions.* Preparatory work and determining feasibility. Attempt evaluation of real-world deep learning model within the DSL. Implement support for intrinsics like linear algebra routines.
  - **Milestone.** Progress Report [February 2].
- **Weeks 3 – 4** (February 1st – February 14th)
  - Prepare slides for presentation.
  - *Extensions.* Implement and evaluate backends using a deep learning framework (PyTorch/Jax) and another using code generation (C/LLVM)
  - **Milestone.** Progress Report Presentation [February 7th – February 14th].

- **Weeks 5 – 6** (February 15th – February 28th)
  - Dissertation – Write Chapter 1: Introduction & Chapter 2: Preparation.
  - *Extensions.* Experiment with optimisations using equality saturation on one of the existing backends.
- **Weeks 7 – 8** (February 29th – March 13th)
  - Dissertation – Write Chapter 3: Implementation.
  - *Extensions.* Investigate and attempt an approach to size types.

D.7.4 Easter Vacation

- **Weeks 1 – 2** (March 14th – March 27th)
  - Dissertation – Write Chapter 4: Evaluation.
  - *Extensions.* Finalise the scope and implementation.
- **Weeks 3 – 4** (March 28th – April 10th)
  - Dissertation – Write Chapter 5: Conclusions.
  - Share full draft with supervisor and directors of studies.
  - **Milestone.** Full dissertation draft shared for review.
- **Weeks 5 – 6** (April 11th – April 24th)
  - Address review comments for draft.
  - **Milestone.** Reviewed dissertation draft.

D.7.5 Easter Term

- **Weeks 1 – 2** (April 25th – May 8th)
  - Editing and proofreading dissertation.
  - Updating dissertation with latest results from extensions.
  - Prepare source code for submission.
  - **Submission [May 10th]**

D.8 Resource declaration

I will use my personal laptop (2021 MacBook Pro, 14-inch) for developing the project. I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure. GitHub will be used to backup the repository and project notes, and Overleaf will be used for the dissertation itself. These are in addition to backups on my other personal devices. All libraries and packages used will be open-source. As part of extensions, I may make use of an external computing cluster, such as the freely available Google Colab or department resources.